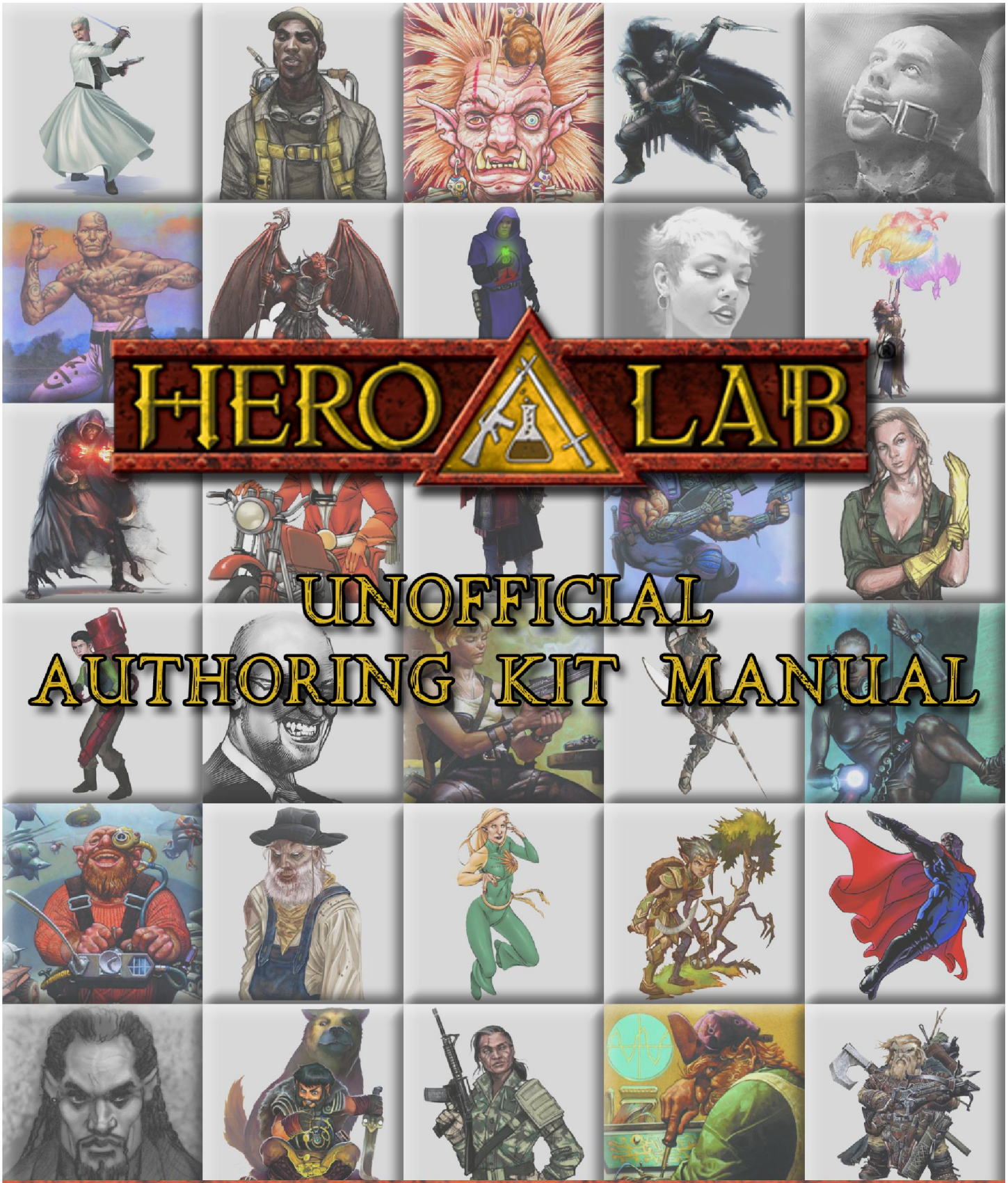

HERO LAB

UNOFFICIAL AUTHORING KIT MANUAL

Edited by R. William Morris

based on the original Authoring Kit Wiki at http://hlkitwiki.wolflair.com/

Table of Contents

Welcome to the Hero Lab Authoring Kit

The Authoring Kit for Hero Lab provides a vast array of capabilities, and those capabilities will continue to evolve with the product. As such, we needed a means of documenting all those capabilities that could readily adapt and evolve as well. We concluded that the best way to accomplish this is to create a wiki that we can extend on an ongoing basis. As an added bonus, the wiki can also enable users to share tips and suggestions.

If you are not familiar with wikis, you can think of them as an intelligently structured assortment of web pages. For information on using this wiki, please refer to the User's Guide (http://meta.wikimedia.org/wiki/Help:Contents).

Wikis are designed to be easily searched, so you can enter whatever term you are interested in and quickly find all the various entries pertaining to that topic. This will be invaluable as you become proficient with the Authoring Kit and want information on specific capabilities. Until you reach that point, you can simply start with this page and follow the various links below to read through all the various topics.

Introduction to the Kit

The goal of the Authoring Kit is to provide everything you need to create and/or edit data files for Hero Lab. When adding material to an existing game system, you can typically utilize the integrated Editor within Hero Lab. However, if you want to create data files for a new game system, you can use the information provided in the Authoring Kit to achieve that objective.

Hero Lab is the first tool that offers a versatile enough engine to handle all the complexities of virtually every RPG system. There is no practical way to develop an engine that inherently handles all this complexity without putting a substantial amount of power and flexibility in the hands of the data file author. With that power and flexibility comes a great deal of material to digest, though, which can make the Authoring Kit seem daunting at first.

To compensate for this, we've invested a great deal of time and effort to simplify and streamline everything, which extends well beyond just refining the engine and how everything works. The documentation is structured to introduce you to concepts in an incremental fashion and make it easy to find the information you need. We've provided a fully operational set of data files as a starting point for your own projects that can be readily adapted to any game system. We've even included a complete walk-through that details how to transform the starting data files into a full-fledged implementation of the Savage Worlds game system.

It's our sincere hope that you'll find the Authoring Kit reasonably straightforward to use and that we've made it possible for you to create quality data files for all your favorite games.

Documentation Conventions

For brevity, the Authoring Kit will often be referred to as simply the "Kit" and Hero Lab will typically be referred to as "HL".

Within the Kit documentation, there are a few conventions utilized. Anytime those important points arise within the text, they will be flagged appropriately using one of the techniques below.

WARNING!	Used to flag the most critical items, which have significant impact on the usability and maintainability of your data files
IMPORTANT!	Identifies considerations that impact the way in which you create your data files, but the results won't usually be horrible if you ignore them
NOTE!	Indicates topics that may influence your choices in data file creation in some situations

PDF Available

The Kit documentation is also available in PDF format for offline reference. The PDF version requires Adobe Acrobat Reader to view. You can download the PDF document via the link below.

http://www.wolflair.com/download/hp/hl_kit.pdf (last updated 23-Feb-2009)

A shorter version of the PDF, omitting the entries from Category: Authoring Examples, can be downloaded here: http://www.wolflair.com/download/hp/hl_kit_noauthoring.pdf (last updated 23-Feb-2009)

IMPORTANT! The PDF version will usually not be as up-to-date as the online wiki, since the wiki is updated on an ongoing basis and the PDF version is updated only periodically. If there is a discrepancy between the PDF and the wiki, always treat the information here in the wiki as the most accurate.

Data File Authoring Topics

Each of the topics below will take you to detailed documentation on the corresponding facet of the Kit. A brief summary of each section is provided below as well. It is highly recommended that you start with the first topic in the list and work your way downward, just like you would normally read the chapters of a book in the sequence they appear within the book. Most chapters build upon the material from previous chapters, so skipping material may lead to some level confusion.

IMPORTANT! The Authoring Kit documentation is a work-in-progress and will continue to be expanded as the capabilities of the product continue to evolve. We've mapped out an extensive long-term plan for both the product and the documentation, and the general structure can be seen herein. However, many sections have not yet been written. These sections will appear as red links throughout the documentation and will be added over time to complete the documentation.

Basic Concepts and Terminology

This section covers all of the fundamental topics that the Authoring Kit is built upon. It's critical that you are familiar with all of these topics before continuing with the other sections.

Advanced Authoring Concepts

Building on the basic concepts, this section outlines more sophisticated mechanisms that you will likely want to leverage.

Kit Reference

Details regarding the syntax and structure for every facet of the Kit are spelled out in this section.

Authoring Examples

This section provides concrete examples showing how to add a wide range of features to your data files. This includes a walk-through that details creation of a complete set of data files for the Savage Worlds game system.

Techniques and Solutions [TBD]

Hero Lab offers a vast array of different capabilities, with different features being appropriate for different game systems. This section provides a laundry list of how to integrate the various mechanisms to tailor your data files to a particular game system.

Skinning the Interface [TBD]

The Kit provides you with the ability to completely change the visual look and feel of your data files. Once the basic functionality is in place, you can adapt the visuals however you like, just like has been done for the games Mutants & Masterminds and World of Darkness.

User Tips and Suggestions [TBD]

This section outlines an assortment of tips and suggestions that have been submitted by other users.

Legal Details

Contact Information

Company Website: www.wolflair.com (http://www.wolflair.com) Technical Support Email: support@wolflair.com (mailto:support@wolflair.com) Discussion Forum: support.wolflair.com (http://support.wolflair.com)

Basic Concepts and Terminology

This section identifies all of the fundamental concepts and terminology that form the backbone of the Kit. A brief summary is provided for general topic below. Click on a topic to get all of the details.

Glossary of Terms
There are numerous terms used throughout the Kit, and they are generally introduced in an incremental fashion. However, if you want one place as a central location for all terms, this is it.

Unique Ids
Just about everything within HL data files is assigned a unique id that serves to uniquely identify that object throughout the data files and enables it to be referenced by other objects. There are specific rules for unique ids that must be adhered to.

XML Files
All the data files you'll work with subscribe to the XML standard. A brief overview of XML files, as they pertain to the Kit, is provided in this section.

Structural Building Blocks
On a structural level, the Kit relies on an assortment of objects that serve as building blocks for everything pertinent to a given game system. These building blocks are outlined in this section.

Visual Building Blocks
Distinct from the structural building blocks for a game system, the Kit has a separate set of objects upon which the visual behaviors of the data files are built. The visual elements are outlined in this section.

The Physical Files
There are an assortment of different file types involved in a complete set of data files. They must reside in specific locations for HL to properly find and use them, and there are critical naming conventions that they must subscribe to.

Data Manipulation Basics
A significant part of data file creation is manipulating the various building blocks through scripts and other related mechanisms. This section details the basics to provide valuable context before delving into them in-depth in the following sections.

Manipulation of Visual Elements
With the vast diversity of RPGs, the author needs to decide how information is presented to the user and how it responds to the user. The manipulation of the various visual elements is a critical element within any set of data files.

Data File Development Process
Now that you've been introduced to all the basic pieces, it's time to look at how everything comes together. The overall process of developing data files is outlined in this section.

Glossary of Terms

The Kit has an array of terminology that you'll need to become familiar with. Most of the fundamental terms are defined below. Other terms are defined as they are introduced within the documentation. The terms below are those that you will likely come into contact with in a variety of contexts during data file creation. Please familiarize yourself with all of these terms, as the rest of the Kit documentation assumes you are familiar with them.

This section provides merely a quickly summary of each term. You'll find more in-depth information on most of these terms in subsequent sections of this documentation.

unique id	Just about everything within the data files is assigned a unique id (often shortened to id) that serves to uniquely identify that object throughout the data files. Each object can then be referenced by other objects based on that unique id.
hero	A hero is the character that the user is creating and evolving within HL.
portfolio	A portfolio represents the collection of heroes being actively created by the user. In many cases, a user will only be creating a single character, but the portfolio can encompass dependent's, henchmen, allies, an entire adventuring party, etc.
thing	A thing encapsulates all of the characteristics of an object the end-user manipulates for a particular game system. These include classes, spells, weapons, skills, etc.
field	Each thing can have an assortment of values associated with it. Each of these values is a field, with some being fixed and others changing dynamically in response to user actions during hero creation.
component	All things can be grouped into sets of related behaviors (e.g. weapons, armor, spells, etc.). Each of these groupings is a component, with each component having a pre-defined set of fields that describe all instances of that component. Every thing is an instance one or more components.
component set	A component set represents a collection of one or more components. Each component corresponds to a particular aspect of the game system, and component sets allow those components to be blended into useful combinations. For example, a game system might have separate components for armor and cyberwear, but some cyberwear might also be armor, in which case a component set could be defined for cyberarmor that blends the two components into a single set.

pick	When a thing is added to a hero, an instance of that thing is created, and that instance is called a pick (because it will typically have been picked by the user). Since it is possible to add multiple instances of the same thing to a hero, and each instance can be configured differently, a separate term is needed for an added thing.
entity	Some things are complex and highly customizable by the user (e.g. vehicles, magic weapons, etc.). When that occurs, an entity is used to describe the customizations that can be performed within a separate level.
gizmo	Like picks, when an entity is added to a hero, an instance of that entity is created, and that instance is referred to as a gizmo. For example, multiple custom magic weapons can be added to the hero, each configured differently, with each being a distinct gizmo and all being based upon the same entity.
container	Both heroes and gizmos hold picks. When no distinction is needed between a hero and a gizmo, they are generically referred to as a container.
portal	Both on the screen and within character sheets, a portal represents a distinct visual mechanism for presenting the contents of a field or some other information to the user. Some portals will accept user input and others coordinate the presentation of large set of data through tables.
style	The visual look of a game system will be standardized so that all portals of a given type will share common characteristics like color, font, etc. To eliminate the need for re-defining this information for every portal, a number of styles are defined and every portal then refers to a style for its characteristics.
template	Portals are grouped into templates to describe how all of the information for a thing should be presented to the user. By defining labels and appropriate visual components to show the contents of fields, a template defines the way things are viewed by the user.
layout	When presenting information to the user, templates and individual portals will need to be combined, and this is achieved through layouts.
panel	On the screen, all information is organized for presentation to the user within panels, with a different tab corresponding to each tab across the top of the main window.
sheet	A sheet corresponds to a character sheet being output for the hero, usually to the printer.
table	Throughout the user-interface, the Kit uses tables to present a scrollable list of templates, wherein each item in the table presents the contents of a single thing or pick from a related set. Tables can sometimes support user-selection as well, in which case a separate selection table is used to choose from.
script	The Kit provides a simple programming language through which you can define scripts that are allow you to implement the nuances of every game system.
phase	Evaluation of scripts, rules, and other operations must be scheduled within the engine to occur in a controlled sequence. To this end, each game system defines a set of phases that break up the sequence into logical chunks.
priority	Within each phase, priorities are assigned that control the sequence within that phase.
eval script	Whenever the selection of a thing causes chain reactions to occur within the hero (e.g. a magic item that increases the wielder's strength), an eval script is written and attached that performs the necessary actions.
eval rule	Every game system has an assortment of constraints that govern hero construction, such as minimum and maximum limits, allowed combinations of abilities, etc. To enforce these constraints, eval rules allow these requirements to be validated.
task	Each eval script and eval rule is scheduled as an individual task within the HL engine for processing.
procedure	A procedure is a fragment of a script that is defined separately and can be re-used from within multiple different scripts. For example, if a game system has multiple pieces of equipment that trigger the same basic set of adjustments to the hero, a procedure could be used to put all the logic in one spot and re-use it.
tag	Each individual characteristic of a thing is identified via a tag associated with that thing. Each tag represents an individual facet of a thing, such as the level of a spell, whether a weapon is ranged or not, etc.

tag group	While individual tags identify a specific characteristic of a thing, a tag group identifies a collection of related tags. For example, spells in the d20 System game system each belong to a particular "spell level", and each spell could be assigned the tag "1", "2", "3", etc. from the tag group "spelllevel".
tag expression	Things will often have many different tags assigned to them. To identify the things that meet a select set of requirements, various tags will be tested for via a Boolean expression referred to as a tag expression (or tagexpr for short).
version	The term version refers to the version of a set of data files (or possibly the Hero Lab product). In order to keep track of updates to data files, a version number is assigned to the data files, which is used both by the end-user and by HL to track when changes are made.
element	Since the Kit uses XML for its underlying file structures, and the term element has an established meaning within XML, this term is used to reference objects that are implemented as XML elements within data files.
attribute	Similar to the use of the term element, the term attribute has an established meaning within XML, so this term is used to reference anything that is implemented as XML attributes within data files.
bootstrap	There are times when one thing or entity needs to automatically add additional things to the container, such as the special abilities conferred for a particular race. These automatically added things are considered to be bootstrapped by the object that adds them.
live	Both structural elements (e.g. picks and gizmos) and visual elements (e.g. panels and layouts) are considered to be either live or non-live, depending on whether they satisfy various requirements. When live, an object behaves normally. When non-live, an object behaves as if it does not exist (i.e. was never added to the container or is never shown to the user).
linkage	Logical associations between one thing and another can be established via linkages. This allows a script to be written for a component that is then inherited by all derived things, but with each individual thing defining a potentially different thing as its associated linkage.
dossier	An ordered collection of character sheets is referred to as a dossier.

UNIQUE IDS

Most objects within HL are assigned a unique identifier so that they can be easily referenced throughout the data files. For example, each thing needs to have a unique value. HL uses unique ids for naming objects. Unique ids must comply with a number of rules, as outlined below. These rules enable HL to achieve significant performance optimizations at run-time, so these restrictions are important.

1. Unique ids can be a maximum of ten (10) characters in length.
2. Unique ids may only contain the standard alphabetic characters (A-Z and a-z), numeric characters (0-9), and the underscore ('_').
3. Unique ids are case sensitive ("foo" is not the same as "Foo" is not the same as "FOO").
4. While it is technically legal for a unique id to start with a numeric character (0-9), it is generally NOT a good idea to do so. If a unique id begins with a numeric character, it will not be able to be used in most scripts, rules, and tag expressions (since starting with a numeric character causes Hero Lab to think the unique id is a numeric value).
5. All unique ids must be unique within a defined context. For example, it is not valid to have two different tag groups with the unique id "mygroup". However, it IS perfectly reasonable to have a tag group named "mine" and a rule set that is also named "mine", since they represent different contexts.

NOTE! The notable exception to #4 above is for tags, where purely numeric ids are common (to represent ranges, levels, resource costs, etc.). Since tags are always referenced with their tag group, a purely numeric tag id incurs none of the liabilities mentioned above.

XML FILES

All HL data files are stored as XML files, as are saved portfolios and most other files intended for user access. Consequently, you'll need to be familiar with the structure of these files in order to manipulate them appropriately. This section outlines the details you'll need to know for this purpose.

If you are not already familiar with XML, it is quite easy to learn, since XML uses simple text files that can be easily created or modified. For additional information on XML, there have been countless books published on the topic and there are extensive resources available on the internet. The official site can be found at the following link. http://www.w3.org/XML/

HL utilizes only the basic mechanisms of the XML standard, so HL files are quite simple to work with. Since XML and HTML both derive from the same set of standards, anyone even tacitly familiar with HTML will be able to pick up XML very readily – at least to the complexity level employed by Hero Lab (or the lack of complexity, actually).

There are numerous commercial, shareware, and freeware tools available for easily editing XML files. Each has advantages and disadvantages, so you will need to make the determination of which tool is "better" for yourself. It's also perfectly reasonable to edit XML files with a simple text editor, although you'll want an editor that at least offers line numbers, since errors are reported with line numbers to allow easy correction of problems.

BASIC XML TERMINOLOGY

You are assumed to be familiar with XML before attempting to write data files for HL. However, it's quite likely that you may be reviewing this documentation before deciding whether you want to try your hand at writing data files, in which case you might not know XML yet. So we've provided very basic definitions of a few fundamental XML terms below to help you better understand the Kit documentation:

- **Element:** HL data files are comprised of XML elements that define all of the information for a particular game system.
- **Attribute:** When creating data files, almost all information is conveyed through the use of attributes within the XML file format. Each XML element contains an assortment of zero or more attributes, where each attribute defines a specific piece of information about the element.
- **PCDATA:** When a block of free-form text is required for an element within Hero Lab data files, that text is specified via the use of XML PCDATA. When using PCDATA, remember that you must enclose the entire text within a CDATA block as a wrapper if you utilize any of the XML reserved characters (see below).
- **CDATA:** If you need to include special formatting and/or reserved XML characters within a PCDATA region, you will want to wrap your text within a CDATA block. A CDATA block simply prepends the text with the character sequence "<![CDATA[" and terminates the block with the sequence "]]>". The list of reserved XML characters is defined further below.
- **DTD:** Every XML file must be assigned a formalized structure for its contents. The formal specification of an XML file's structure is via a DTD (short for Data Type Definition). An appropriate DTD should be included with this documentation for every public HL file format.

RESERVED XML CHARACTERS

The XML language has a number of reserved characters that have special meaning. If you need to use these characters within your data files, an appropriate replacement must be used in accordance with the XML language specification. Alternately, a CDATA block can be used within a PCDATA region. These special characters are documented in detail within any reference on XML, but they are repeated below for convenience. Any time you need to use one of the characters below within HL files, use the corresponding character sequence given on the right.

<	&lt;
>	&gt;
&	&amp;
"	&quot;
'	&apos;
literals	If you need to specify a character with a code of 128 or higher, you need to specify the character as a literal. The syntax for this is "&#ddd;" where "ddd" is the decimal value of the character code (e.g. "•"). You can also specify hexadecimal values via the syntax "Ý" (e.g. "®").

XML COMMENTS

If you are editing data files by hand, then you can freely insert comments into the XML data files using standard XML syntax. Comment blocks begin with the character sequence "<!—" and end with the sequence "—>". Any number of lines of text with any contents can appear within an XML comment block. Comments within XML files may not be nested.

For example, the following XML includes a comment that effectively omits the "dropped" element from the document, including all of its attributes and the "dropchild" child element as well.

```
<document>
 <first attrib="value"/>
 <second attrib="value" another="junk">
  <child>This is PCDATA</child>
<!--
  <dropped attr1="x" attr2="y" attr3="z">
   <dropchild attr="ignore"/>
  </dropped>
-->
  <third dummy="nothing">
  </document>
```

NOTE! If you create a data file outside of HL and then use HL's integrated Editor to edit the file, all comments will be thrown away by the Editor. When using XML comments, be sure to only edit those files within tools that will preserve the comments.

XML CHARACTER ENCODING SETS

The XML specification identifies a number of character sets that can be utilized within a given document. Unfortunately, none of them fully support the Windows ANSI character set, and HL is a Windows application. HL assumes all XML documents subscribe to the XML character encoding set that most closely approximates Windows ANSI, and all characters within that set are assumed to be the corresponding Windows ANSI characters. This means that HL assumes all XML documents utilize the "ISO-8859-1" character set (more commonly referred to as Latin-1), with a number of exceptions that are detailed in the Kit Reference section of the documentation.

The identity element at the top of all XML files should specify an encoding of "ISO-8859-1" for completeness. If no encoding is given, ISO-8859-1 is assumed. An example is given below:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

NOTE! There is an unofficial XML encoding named "Windows-1252" that properly reflects the Windows ANSI character set and is often used. However, various XML parsers do not recognize this encoding set due to its unofficial nature. In the interest of maximum compatibility, the modified Latin-1 set is used instead.

STRUCTURAL BUILDING BLOCKS

OVERVIEW

In order to support all RPG systems, HL has distilled out a collection of generalized building blocks that can be used to construct the mechanics of each game system. The engine leverages these flexible, fundamental building blocks in a highly object-oriented fashion. For example, every game system object (spell, feat,

skill, class, etc.) has its distinct structure defined, and every individual object is a specific instance of the appropriate type. Each game system defines its own unique set of object types and all of the individual objects of each type. For example, the data files for the d20 System define object types for classes, feats, skills, spells, etc., while the data files for the World of Darkness game system define object types for clans, covenants, merits, disciplines, etc.

The data files for each game system dictate the set of object types available for that game system, as well as the set of objects for each of those types. The Kit provides an assortment of pre-defined building blocks that you can use, adapt, replace, and extend. The net result is an extensible framework that provides a substantial set of core functionality and allows significant customizability for authors to adapt to virtually any game system.

Heroes, Actors, and Portfolios

Heroes and Actors

The fundamental purpose of HL is to facilitate the creation and management of characters for role-playing game systems. Within HL, individual characters are typically referred to as "heroes", although each game system can define an alternate term for display to users if appropriate. Since the term "hero" can be renamed, and since "hero" isn't really a suitable term when creating villains, monsters, and other types of NPCs, the Kit often uses the term "actor" instead.

There is no difference between the terms "hero" and "actor" within the Kit. They are interchangeable.

Portfolios

HL saves actors within a "portfolio". Multiple actors may be saved within the same portfolio.

Each portfolio is self-contained and makes no references to any external files. If an actor has a character portrait image file associated with it, the image is thereafter managed and stored within the actor and portfolio, with no further connection to the original file.

Components and Component Sets

Each distinct type of game system object is defined via a unique component set (often shortened to "compset"). The component set defines a specific set of characteristics for an object type. For example, in the d20 System, skills have a particular set of behaviors (e.g. the number of skill points assigned, the linked attribute, etc.), while weapons have a different set of behaviors (e.g. damage type, damage value, critical score, etc.). A separate compset is therefore defined for each type of object you want to create.

Compsets get their name because they are actually a set of one or more components. Various object types will have similar behaviors to each other. For example, all types of equipment have a cost and weight, whether the equipment is a weapon, armor, magical rope, or merely a wineskin. All of the details associated with the cost and weight of equipment should ideally be handled in a simple, consistent fashion. By defining a single component that handles everything to do with cost and weight, and re-using that within the various compsets for each equipment type, the Kit becomes powerful, flexible, and efficient for the data file author.

Through the intelligent design of components and component sets, you can re-use common logic across multiple similar object types. The Kit provides a number of standard components and compsets that you can readily build upon and/or adapt to the specifics of your game system.

Things

Each specific game system object is always based upon a specific compset. These objects are referred to generically as "things" within Hero Lab. For example, the "Longsword" thing might be based upon the "Melee" compset, while the "Fireball" spell would be based on the "Spell" compset.

The compset dictates the general behaviors of all individual things that are based upon it. However, each thing then tailors its own nature and behavior, as appropriate.

All things possess a variety of basic characteristics that are inherently unique to each thing, such as a name and description. Thing also have various characteristics that are inherited from the components from which the thing is derived (which are dictated by the compset). Lastly, things have an assortment of special characteristics that can be optionally specified to customize their individual behavior, including scripts, rules, and a host of other facets (all described elsewhere in this documentation).

Some of you out there in reader-land might be wondering why we used the term "thing" instead of the more standard term "object". The reason is that there are other facets of the underlying HL engine and the Kit that are also very object-oriented in nature. We wanted to be able to use the term "object" in the documentation to refer to general object-oriented behaviors, without specifically linking the word to this one situation. Using the term "thing" for compset-derived objects gives us that ability.

Fields

Fields represent values that can be assigned to things. For example, a weapon might have a field to specify the damage it does, while a skill would have a field to track how proficient the character is with that skill. Fields can be specified as being either numeric or text-based in nature, with text-based fields requiring that a maximum character length be designated.

Fields are defined within components. As such, a given field is inherited by every compset that incorporates the component it is defined within. This means that the fields possessed by a thing are dictated by the compset it is based upon, with all the fields defined within the compset's constituent components being possessed by the thing.

Every component can define zero of more fields. No fields are actually required for a component, and there are times when using no fields are appropriate, but most components will be defined with at least one field.

Field Values

Fields can be assigned a default value when defined within a component. This value will be inherited as the default value for every thing that includes the field. When defining a thing, you can either inherit the default value for a field or specify the appropriate value to use for it.

The values of fields can be dynamically modified at run-time in response to events triggered by user selections. Many fields will be wholly derived at run-time, such as a character's attack skill with a particular weapon, which must be calculated from all the various

factors that contribute to the final result. This is achieved via the use of scripts.

The Kit also supports the definition of fields as both arrays and matrices. If this is done, the maximum dimensions of the array or matrix must be specified and HL will allocate storage for all elements in the array/matrix. Further details on arrays and matrices are defined as an advanced concept.

NOTE! Any attempt to assign a text value longer than the defined maximum to a given field will result in the text being truncated at the maximum length.

Tags and Tag Groups

At the heart of HL's engine is an important concept that is used as one of the basic building blocks for data files: tags. Things can be assigned a virtually limitless variety of tags, where each tag represents a label that describes some facet of the thing. Tags are managed in sets referred to as tag groups. All of the tags in a given tag group are intended to have a related meaning, such as "race", "level", "color", etc. The set of tags and tag groups is unique to each game system.

Things can have any number of tags assigned from any number of tag groups. Sometimes you'll use zero tags and sometimes you'll use a dozen or more for a complex set of behaviors. You can also freely re-use tags on different types of things, and there are times when using the same tags on different types of things can be extremely powerful. One key advantage of tags is that they are fast and flexible when compared to fields. Modifying a field requires modifying the underlying component and then adjusting all of the things derived from that component, whereas tags can be added, modified, or deleted at any time. The most powerful advantage of tags, however, is that tags can be easily utilized in tag expressions, which are a powerful and flexible mechanism used extensively within data files (and discussed in detail elsewhere).

Referencing Tags

Since tags always belong to a given tag group, an individual tag is always identified with the tag group to which it belongs. This is critical, because the same tag could be used in two or more different groups, with each having a different meaning. For example, the tag "Common" could have two very different meanings in two different tag groups for some game systems. Within the "Language" tag group, "Common" might refer to the "Common tongue" used in the game world. However, within the "Rarity" tag group, the "Common" tag might indicate that an item is commonly available for purchase instead of being uncommon or rare.

When referring to tags, the notation used is called a tag template. This notation specifies the unique id of the tag group, followed by the '.' character, followed by the unique id of the tag. For example, the reference "Language.Common" would refer to the "Common" tag within the "Language" tag group. Remember this syntax, since tags are used extensively and you'll be seeing the tag template notation frequently.

REMINDER! Unique ids for almost all elements must be globally unique. The key exception is with tags, for which the unique id must be unique only within the containing tag group. This is possible because tags are always referenced with the associated tag group, so uniqueness is always ensured, even though two tags in different groups may have the same id.

Using Tags

In most cases, you'll be assigning a single tag from a given tag group to a thing. However, sometimes you'll find that the lack of a tag from a particular group can have important meaning. At other times, you'll find that you need to assign multiple tags from a given tag group to a particular thing. For example, spells in the d20 System have different casting behaviors (e.g. verbal, somatic, and material), with any number applying to a particular spell. In this case, you would assign the appropriate tags to a given spell, with the presence of a tag indicating the behavior and the absence of the tags indicating the opposite.

Dynamic Tags

In most cases, each tag group will explicitly define the complete set of tags that exist for that tag group. However, there will be times when a tag group needs to be open-ended for expansion. For example, consider the "Language" tag group mentioned above. You could populate the group with all of the languages in the game system, but you can be sure some designer will come along at some point in the future with a new language for some new race. To handle these situations, a tag group can be designated as dynamic, allowing new tags to be defined on-the-fly as they are needed. When you define a new dynamic tag, you must specify all its various details.

Tag Values

Tags will usually be used to identify a non-value attribute of a thing. Once in a while, though, tags will be used to convey a value in addition to an attribute. For example, each spell within the d20 System has a spell class and a level for that class (e.g. Wizard level 4, Cleric level 2). Since the same spell can apply to different classes at different levels, it would be difficult to track all that information easily. So one solution would be to define a "SpellLevel" tag group with tags that combine the class and the level, such as "Wiz4" or "Clr2". The data files could then identify a spell's class from the prefix on the tag and also get the value from the suffix. The tag value of a given tag is the determined by extracting the trailing digits from the tag's unique id and converting it to an integer value. So the tag id "xyz123" would have the value "123", while the tag "a1b2c3d4" would have the value "4" (the extraction stops at the first non- digit encountered, starting at the end and working towards the front). If a tag id does not end in any digits (e.g. "black"), the tag's value is always treated as zero.

Tags Are Often Better Than Fields

The concept behind fields will likely be familiar to you, while the concept behind tags may not. This makes it quite possible that you will be inclined to use fields to solve most problems. However, you will benefit from learning how to leverage tags, since there are many situations where tags are a better solution than the use of fields.

Tags provide a fast and efficient means of assigning an attribute to something. Tags are ideal when valid values consist of a pre- defined set. For example, spells in the d20 system have a level, a range, an area of effect, and some combination of three different casting behaviors: verbal, somatic, and material. The range and area of effect have a very wide range of possible values, so a field is probably the best way to manage that characteristic. However, the spell level and casting behavior are ideal situations for a tag. There is a fixed number of spell levels and there are only three casting behaviors. Two tag groups could be easily defined to solve this (one

for spell level and one for behavior), and individual tags could be defined for each level and each behavior. Appropriate tags would then be assigned to each spell corresponding to the spell level and casting behaviors.

A field could also be used for each of these, but tags are a better solution. This is especially true for the casting behavior, where using a field would either entail having a numeric code that is not immediately obvious (e.g. 1=verbal, 2=somatic, etc.), or it would require using a text value (e.g. "verbal", "somatic"). Using a text value would be prone to data entry typos over the span of hundreds of spells. In contrast, the use of the tag "verbal" makes the value obvious and the requirement that all tags match an existing set eliminates the possibility of typos.

PICKS

Consider a situation where an actor possesses two separate longswords. Both of these weapons correspond to the exact same thing that has been defined within the data files. Now consider what happens when a magical effect is applied to one of the longswords, but not the other. Similarly, consider a customizable weapon, such as a modern assault rifle, where one rifle might be configured with a laser scope and infrared optics, and a second rifle might be given recoil compensation. In each case, both weapons are based upon the same underlying thing, but the weapons must be kept distinct from one another and modified independently.

To handle situations like this, whenever a thing is added to an actor, it becomes a completely independent object. This new object is referred to as a "pick". Each pick references the original thing and inherits all of its characteristics as its initial state. However, all dynamic changes are applied independently to each pick.

IMPORTANT! Throughout the Kit documentation, the terms "thing" and "pick" will be used extensively. It is critical that the distinction between the two be clearly understood. Otherwise, the Kit documentation could become confusing, since every behavior that can be applied to things can also be applied to picks (i.e. picks are essentially a superset of things).

ENTITIES AND GIZMOS

Things and picks work for most situations, but they just aren't sufficient when you need to support something complex, such as a customizable vehicle or magic item. In situations like this, you'll need to define something that can itself contain different picks, thereby allowing for customization by the user.

The Kit utilizes an "entity" for this purpose. Each entity can be specified as starting out containing various things and/or tags, thereby allowing you to pre-configure various facets of the entity.

An entity is always attached as a child of a thing. A given thing can only attach a single entity, and a thing assigned a child entity will always attach that entity whenever it is added to an actor.

When a pick is added to an actor and that pick has a child entity, the new entity is referred to as a "gizmo". The distinction between entity and gizmo parallels the distinction between a thing and a pick. The entity is the definition, but the gizmo is the actual instance that has been added to the actor. Since it is possible to attach multiple instances of the same thing to an actor, and that thing can have a child entity, it's also possible to have multiple instances of the same child entity. Each of those is a distinct gizmo.

When added to an actor, each gizmo inherits its starting state based on the entity. If the entity is assigned various tags, those tags start out on the gizmo. Similarly, if the entity is assigned starting things, each of those things is added to the gizmo as a pick within that gizmo.

CONTAINERS

The term "container" is used to generically refer to both actors and gizmos, since both actors and gizmos contain an assortment of picks. Actors involve a great deal more behaviors than gizmos, and gizmos are not a proper subset of actors. However, whenever something applies equally to both actors and gizmos, it will be described as applying to containers.

MINIONS AND MASTERS

Many game systems involve the creation of actors that are secondary to a character. For example, hirelings and henchmen are a common situation. In the d20 System, wizard familiars and druidic animal companions are tied to the character in a secondary relationship. In Mutants & Masterminds, sidekicks and minions are purchased out of the character's available power points.

To support these various situations, the Kit provides a generalized mechanism to create and maintain a hierarchy of actors. This hierarchy consists of "masters" and "minions", wherein an actor is the master of any actors beneath it in the hierarchy and a minion of any actor above it.

In the same way that an entity can be defined and associated with a thing, so can minions. The key difference is that minions are always actors, and HL already knows how to create an actor, so there is no need to define a separate "minion" element. All that is needed is to associate a minion with a thing and specify any special characteristics for the minion. Whenever that thing is added to a character as a pick, a new minion is automatically created and customized appropriately.

The actor hierarchy does not have a maximum depth. This is necessary to handle situations where a character has a minion, and that minion has its own minion, etc.

LEADS

Not all actors are created equal. As such, their relationship to one another is critical in determining a few subtle, yet important, distinctions between actors. In a nutshell, a "lead" is any actor that is not a minion of any other actor (i.e. top-level).

Whenever a new portfolio is created by the user, an initial character is always created as well. This new actor is considered to be a "lead" actor. Similarly, whenever the user creates a new actor in an existing portfolio by means of the "Create New Hero" option on the "Portfolio" menu, the new actor is considered to be a "lead". And the same applies to any new actor that is directly imported from another portfolio, including a stock portfolio.

The exact implications of leads versus non-leads are described in the relevant sections elsewhere in this documentation. However, the distinction is important, so it is set forth here.

USAGE POOLS

Most of the time, the information managed for a character simply involves tracking the current state. However, there are times when a history of changes needs to be maintained. To handle these situations, the Kit provides a mechanism referred to as a "usage pool".

Usage pools track both the current net value for the pool and a history of adjustments applied to the pool, up to a configurable maximum history size. A usage pool can be defined for an actor or for an individual pick. This makes it possible to use a usage pool to track the overall experience and cash accrued for an actor, as well as for tracking the individual experience and cash accrued for a single journal entry.

Usage pools are also invaluable for tracking damage and similar resources. Since a history is maintained, it becomes possible to easily undo damage that was sustained. It's also possible to show the history to the user for review, such as the sequence of damage and healing that took place during a lengthy combat.

VISUAL BUILDING BLOCKS

The visual presentation of each game system is wholly distinct from the underlying mechanics of the game system. However, it is no less critical to the overall usability of the data files. All the visuals are constructed from an assortment of generalized building blocks that are designed to allow you to tailor the interface to the nuances of each particular game system.

The overall structure of the visual interface is dictated by the way HL is designed. For example, the use of edit panels, summary panels, and tables of information will be common across every game system. Within that overall structure, though, the data file author can tailor the interface in a wide range of different ways.

In general, you will find yourself designing the contents of panels and sheets, which are very similar in how they work. Both contain one or more templates and/or layouts, which allow you to position groups of individual visual elements. Templates contain one or more portals, while layouts contain a combination of portals and/or templates. This general structure applies to all visual elements that you'll create as an author.

The Kit provides a variety of pre-defined, visual building blocks that you can use, adapt, replace, and extend. This should make it possible to get something working in relatively short order, while also allowing you to extensively customize the interface to the needs of virtually any game system.

RE-USING VISUAL ELEMENTS

Visual elements are all globally defined. Consequently, you can define a template once and re-use it in multiple tables or layouts. Similarly, you can define a layout once and re-use it in multiple panels or sheets.

However, what if you want to display multiple pieces of information that share the same template from within the same layout? Or what if you want to re-use the same layout within the same panel or sheet? The Kit uses template references and layout references when specifying the use of these visual elements. Multiple references can re-use the exact same template or layout, with each reference being assigned a different logical name for use within the containing visual element. The logical name is then used in any positioning scripts for the containing visual element to uniquely identify the correct template or layout.

PORTALS

Portals are the finest level of control within a visual presentation. They provide access to individual fields within things and picks. Fields are hooked up to portals, thereby allowing the user to view and/or modify the contents of those fields. For example, a "label" portal could be associated with the "damage" field for a weapon, thereby allowing the contents of the field to be displayed to the user via that portal. If an "edit" portal were used instead, then the contents of the field could potentially be modified by the user.

Each of the different types of portals is summarized in the sections below. Complete details on each portal type will be found within the Kit Reference section of this documentation.

IMPORTANT! Portals are unlike most other elements used within the Kit. For portals that are defined within templates, the unique ids of those portals must only be unique within the context of the template. This means that you may freely re-use the same unique id for portals within different templates. For example, you could have a "name" or "delete" portal within multiple templates. Since portals within templates are only accessible through the template in which they are defined, their scope is uniquely restricted to the template, which makes the re-use of ids possible. Please note that this ability to re-use unique ids only applies to portals defined within a layout, so it does not apply to tables and other portals used directly within layouts.

LABEL PORTALS

Labels are the simplest type of portal and allow the display of information to the user. There are different types of label portals, depending on what the label contains and how it is to be displayed to the user.

- **Literal:** Literal label portals display a fixed string of text.
- **Field-Based:** Field-based label portals show the contents of a specific field.
- **Script-Based:** Script-based label portals synthesize their contents via a script, with the results being displayed.
- **Titles:** Title label portals present their contents with special formatting for use as a title.

IMAGE PORTALS

Image portals enable the display of images to the user. There are different types of image portals, depending on what the portal contains and how it is to be presented.

- **Literal:** Literal image portals always display the same, fixed image.
- **Field-Based:** Field-based image portals show the image dictated by the contents of a specific field.
- **User:** User image portals present the image that has been selected by a user.
- **Reference:** Reference image portals allow a user-added image to be dynamically adapted for use at run-time (such as using a tiny version of the first character portrait on the Tactical Console).

EDIT PORTALS

Edit portals enable the user to edit the contents of a specific field. There are different types of edit portals, depending on what the portal contains and how it is to be presented.

- **Simple:** Simple edit portals provide direct editing of a numeric or text field.
- **Date:** Date edit portals offer structured editing that ensures the contents of the field always subscribe to

specific rules (e.g. four digits for year, two digits for month, etc.).

INCREMENTER PORTALS

Incrementer portals provide the ability to edit the contents of a specific numeric field. An incrementer includes visual elements like a "+" and "-" button that allow the user to easily increase and decrease the value shown within the portal. Incrementers are typically used for attribute values, damage tracking, charges, and other similar mechanisms.

CHECKBOX PORTALS

Checkbox portals allow the user to toggle the state of a specific field between values of zero and one. The user sees the portal as an either/or toggle state.

MENU PORTALS

Menu portals enable the user to select one choice from a list of available choices. There are different types of menu portals, depending on what the portal contains.

- **Literal:** Literal menus present a choice from a set of options that are pre-defined and fixed. They are suited for situations like the gender of a character and the method used for selecting ability scores in the d20 System.
- **Array:** Array-based menus use a script or other technique to synthesize the list of valid choices the user can select from.
- **Things:** Thing-based menus dynamically identify a group of things from which the user can select. They are useful when choosing a weapon type or spell type to assign a skill bonus.
- **Picks:** Pick-based menus dynamically identify a group of picks from which the user can select. They are ideal when you need apply an adjustment to something that is restricted to what the character possesses, such as choosing a weapon that the character has.

CHOOSER PORTALS

Chooser portals allow the user to select a thing from a table of options that include full descriptions. Once the thing is selected, it is added to the character as a pick and becomes an integral part of the character. Chooser portals are perfect for selecting a race or an alignment, since exactly one selection must be made for the character, and the user will want to be able to review the details associated with each possible choice.

ACTION PORTALS

Action portals are presented as buttons that trigger a specific action when interacted with by the user. There are many types of action portals, depending on what action is desired:

- **Delete:** Triggers the deletion of a pick from the table that added it.
- **Info:** When clicked or the mouse is moved over the portal, full details on the pick are displayed.
- **Edit:** Triggers editing of the gizmo associated with a pick by bringing up a suitable form for the purpose.
- **Form:** Triggers the display of a specific tab panel as a form, such as is used to directly apply damage to actors from within the Tactical Console.
- **Trigger:** Invokes a script, such as when applying damage to an actor or resetting the accrued damage.
- **Notes:** Displays a form where arbitrary notes can be edited for a pick.
- **Load:** Switches to the actor associated with the specified pick, such as is used on the Dashboard and Tactical Console.
- **Lock:** Transitions the character into a locked state for the purposes of advancement.
- **Unlock:** Transitions the character into an unlocked state for the purposes of advancement.
- **Master:** Switches to the actor that is the "master" of the current actor.
- **Minion:** Switches to an actor that is the "minion" of the current actor.
- **Manage Gear:** Displays a menu through which the user can move gear between different containers.
- **Get Gear:** Displays a menu through which the user can move gear between actors.
- **Start Combat:** Begins a new combat within the Tactical Console.
- **End Combat:** End the current combat within the Tactical Console.
- **New Turn:** Starts a new combat turn within the Tactical Console.
- **Initiative Change:** Incorporates any direct initiative changes made by the user within the Tactical Console.
- **Integrate:** Integrates any pending actors into the current combat.
- **Sort Dashboard:** Triggers a re-sort of the actors shown on the Dashboard.

REGION PORTALS

Region portals make it possible to place a border around a rectangular region that doesn't correspond to a single portal. For example, if you want to visual group a few portals, you can define a region that encompasses all of them and place a border around the region, thereby placing a box around the group of portals.

SEPARATOR PORTALS

Separator portals allow you to put either a vertical or horizontal separator bar between sections of the display. Separators are an excellent way to visual group portals and make the interface more intuitive for the user.

TABLE PORTALS

Table portals present a collection of related picks in a tabular list. Each item in the table is displayed via a template that is specified with the table. The template specifies how to position the various facets of the pick within the table entry. There are different types of table portals, depending on what the portal contains and how it is to be presented:

- **Fixed:** Fixed table portals display a table of picks that provides no means for the user to add or delete items.
- **Dynamic:** Dynamic table portals display a table of picks that includes an option at the bottom to add new items to the table. If the "add" option is used, the user is presented

with a list of things to choose from and the one selected is added to the container as a new pick.

- **Auto:** Auto table portals display a table of picks that includes an option at the bottom to add new items to the table. If the "add" option is used, a specific thing is added to the container as a new pick, which the user can subsequently customize. The journal is an example of an "auto" table, where adding a new item automatically adds a new journal entry.

SPECIAL PORTALS

Special portals are those portals that have highly specialized uses and cannot generally be customized by the author in term of behavior. However, they do need to be properly placed within the interface, so a special portal is used to represent the mechanism and allow placement. There are a number of special portals, as outlined below:

- **Edit Settings:** Displayed as button that allows the user to edit configuration settings when clicked. This portal is typically used on the Configure Hero form.
- **Settings Summary:** Displayed as a grid that shows the currently selected configuration settings. This portal is typically used on the Configure Hero form.
- **Alliance:** Displayed as a menu that allows the user to select whether the character is to be considered an ally or an enemy for the purposes of the Tactical Console. This portal is typically used on the Configure Hero form.

OUTPUT PORTALS

The output of character sheets entails a number of critical differences from the interactive visual elements used on-screen. As such, a different set of portals is used to render character sheet output. The presentation behaviors of output portals are similar, yet distinct, from interactive portals, and the set of available output portals consists of those outlined below:

- **Label:** Renders text onto the character sheet, with the text being driven by any of the different mechanisms supported by on-screen label portals.
- **Image:** Draws an image onto the character sheet, with the image being handled in any of the different ways supported by on-screen image portals.
- **Table:** Renders a table of picks to the character sheet, using a template to specify the contents of each item within the table.

TEMPLATES

Templates contain one or more portals and represent a rectangular region within the containing layout. The template is responsible for coordinating the position of its own portals within its boundaries.

Every template is associated with a specific thing or pick. All of the portals within the template are associated with that thing or pick. Therefore, all fields associated with portals reference that field with the thing/pick associated with the template. For example, if a template is associated with the "pistol" pick and the template contained a portal associated with the "range" field, that portal would display the contents of the "range" field of the "pistol" pick.

IMPORTANT! Since templates are associated with a thing or pick, templates may not contain portals that are precluded from having field associations. This means that table and chooser portals may not be used within templates.

LAYOUTS

Layouts contain one or more visual elements, where those visual elements can be templates and certain types of portals. Layouts also represent a rectangular region within the containing panel or sheet. The layout is responsible for coordinating the position of its contained visual elements within its boundaries.

Layouts make it easy to group related visual elements together and position them as an atomic unit. For example, within the d20 System data files, there is a layout that manages the selection of class-specific special abilities, another layout for viewing the special abilities for the class, etc. By using layouts, new edit panels for custom classes can be quickly constructed by combining the appropriate layouts for the separate features of a given class.

IMPORTANT! Since layouts are not associated with a thing or pick, layouts may only contain portals that do not have field associations. This means that, in general, only label portals, image portals, chooser portals, and table portals may be used within layouts. In the case of labels and images, the field-based versions of those portal types may not be used within layouts, since no field association exists.

PANELS AND FORMS

Panels and forms are the top-level visual element used on-screen. Panels define the contents of pre-defined regions within HL. This includes the various tab-based panels used within the interface, as well as the summary panels shown on the right. Forms behave very similarly to panels and are used to define the contents of standalone windows that HL manages. These include the Configure Hero form, the Tactical Console, and forms for editing the contents of gizmos (e.g. custom magic weapons within the d20 System).

Panels and forms have a few important behavioral differences, which is why they are kept distinct. Besides their usage, the most important difference is their sizing behavior. In the case panels, the HL engine tells the panel what its available dimensions are, after which the panel sizes and positions its contents accordingly within those dimensions. For forms, however, the dimensions are dictated by the form itself. Once the dimensions are specified by the form (via its Position script), the HL engine will construct a suitable window that encompasses the form and display it.

Both panels and forms contain one or more layouts and represent a rectangular region comprising the panel/form. The panel/form is responsible for coordinating the position of its contained layouts within its boundaries.

IMPORTANT! Unlike most other visual elements, panels and forms share the same namespace. This means that the unique ids assigned to panels are **not** distinct from the unique ids assigned to forms. You may **not** define a panel with the same unique id as a form, and vice versa. All of the ids assigned to panels and forms must be distinct from one another.

SHEETS

Sheets are the top-level visual element used for character sheet output. Sheets define the contents of individual pages of printed output. Sheets contain one or more layouts and represent a rectangular region comprising the sheet. The sheet is responsible for coordinating the position of its contained layouts within its boundaries.

In order to handle lengthy output that spans numerous pages, sheets can be designated as "spillover". The HL engine will automatically track which items have and have not been output, allowing the same sheet to be printed continuously with the "same" contents. With each page, only the material that has not been printed is included, and output finally stops after all the material has been output a single time. This makes the output of material like spell lists and journal logs extremely easy to manage.

SCENES

The term "scene" is used to generically refer to panels, forms, and sheets. All of these visual elements manage layouts and have numerous similarities in how they behave. While each has its own distinct characteristics, whenever something applies equally to all three, it will be described as applying to scenes.

DOSSIERS

When HL presents the user with the option to output character information, the list of dossiers is used. This applies to character sheets, statblocks, and even data being exported for use in other products. For character sheets, each dossier is an ordered collection of one or more sheets, which allows individual sheets to be easily re-used across multiple dossiers. For other forms of output like statblocks, dossiers utilize scripts to synthesize the properly formatted text.

RESOURCES

All fonts, colors, bitmaps, and borders used within the data files must be defined as resources. These resources can then be referenced by styles for subsequent use by portals.

In order to make it easy to create data files for a new game system, the Kit includes a large assortment of built-in resources that can be immediately put to use. As such, you can largely ignore the need to define visual resources when developing your data files, allowing you to focus solely on getting the underlying functionality into place first. Once the data files are working the way you want, you can then switch your focus back to the resources and being the process of replacing them with something more suitable to the game system.

The Kit also publishes all of the "system" resources that are used by fundamental HL mechanisms and allows them to be modified. This makes it possible to completely replace the system resources used throughout HL, thereby completely transforming the visual look of the entire HL interface. The Mutants & Masterminds data files provide an excellent example of the extent to which you can transform the HL interface.

STYLES

The Kit makes is easy to manage and tailor the visual look and behavior of the data files through the use of styles. Every portal must specify a variety of colors, fonts, bitmaps, borders, and other facets to be used when displaying that portal. In order to achieve a consistent interface, you'll typically want to use the same basic look and behaviors for most portals of the same type. Similarly, if you want to change the visual look or behaviors of a particular portal type, you'll want to do it for all of them.

To keep things convenient and easy, you'll define styles that encapsulate the common visual behaviors for each portal type. Each portal is then assigned a style that dictates its visual behavior. Changing the behavior for a portal requires simply changing the assigned style, while changing the behavior of all portals of a type requires simply changing the style definition itself.

You can also dynamically change the style assigned to a portal at run-time via scripts. This allows you to customize the visual look of a portal based on conditions determined at run-time, such as flagging something in red that is determined to be invalid.

SORT SETS

The Kit utilizes a "sort set" to appropriately sort the contents of a list of things or picks. Each sort set is essentially an independent sort specification. You can define any number of sort sets and they are not officially associated with any particular lists of objects. As such, you can easily re-use a particular sort set in a variety of situations.

The purpose of a sort set is to spell out the exact rules to be used for sorting a collection of picks or things. Each sort set will consist of one or more sort criteria, and those criteria must be specified in the exact sequence that you want HL to sort the items in.

Each sort criteria consists of a few characteristics. The first aspect is the unique id of either a tag group or a field. The second aspect is whether to sort the items in an increasing order or a decreasing order.

If a tag group is specified, then the sort criterion instructs HL to sequence all items based on the sequencing rules set forth for the tag group. All items that possess a tag from the tag group are sorted based on the tag group's sequencing rules, and any items that lack any tag from the sort group are sorted last.

If a field is specified, then the sort criterion tells HL to order the items based on the results of comparing the values of the designated field. The comparison rules use a simple numeric comparison when the field is a value and a string comparison when the field is text-based. If a particular item does not possess the field for some reason, then that item is always placed at the end of the sorted list.

When multiple sort criteria are specified, they are processed in the order given until one yields a difference, at which point all further sort criteria are ignored.

The starting data files provided by the Kit include a number of pre-defined sort sets. These sort sets provide an excellent starting point and are already used within the starting data files. You can readily revise and extend the set provided for your own needs.

ENCODED TEXT

While not exactly a true "building block", encoded text is an important facet of the visual presentation for any game system. Encoded text is the term used for inserting control over colors, fonts, and even bitmaps within text strings. If you want to highlight a word within a string in red, you'll use encoded text. If you want to change the font size for a few works or put a keyword in bold, you'll use encoded text. You can also insert bitmaps into the text stream with encoded text, which makes it possible to do things like display the sequence of dots for traits within the World of Darkness game system.

The encoding syntax uses the characters '{' and '}' to identify the special codes, much like HTML uses the '<' and '>' characters to wrap special codes. The text found between the '{' and '}' characters is interpreted based on the table given below.

{br}	Inserts a carriage return into the text at this position.

{nbsp}	Inserts a non-breaking space into the text at this position. Word-wrapping behavior is not allowed to occur on a non-breaking space, so the space is tied to whatever other text appears before and/or after it. This can be useful for placing padding around text that changes foreground and/or background colors.
{spc}	From this point forward in the text, all sequences of multiple spaces are collapsed into a single space.
{b} {/b}	Text following the "{b}" sequence is rendered in bold. This persists until the "{/b}" sequence turns off bold rendering.
{i} {/i}	Text following the "{i}" sequence is rendered in italics. This persists until the "{/i}" sequence turns off italics rendering.
{u} {/u}	Text following the "{u}" sequence is underlined. This persists until the "{/u}" sequence turns off underline rendering.
{font name}	From this point forward in the text, the font with the specified "name" will be used for output. The prevailing size and style are used in conjunction with the new font. **NOTE!** Use of a font that is not a standard Windows font will result in the new font being ignored on any computer that does not possess the font. So be sure to only use standard Windows fonts if you plan on distributing your data files to others.
{size value}	From this point forward in the text, the font size is changed to the given "value". In order to support fractional point sizes, the value is the number of quarter-points, so a value of 40 would translate to a point size of 10, while a value of 38 would translate to a point size of 9.5. The prevailing font and style are used in conjunction with the new size.
{revert}	From this point forward in the text, the original default font characteristics are restored. All states for bold, italics, and underline are reset to off, regardless of the default font characteristics restored. The primary use for this encoding is to output some text in the default font, configure the text properly for a short segment, and then revert to outputting the rest of the text in the original font. This way, the same encoded text works consistently even when the default font varies between situations.
{text xxxxxx}	This sequence changes the foreground text color to the color given by "xxxxxx". The format for the color uses standard HTML color syntax, with each character representing a hexadecimal digit. The first two characters define the Red color value, the next two Green, and the last two Blue. For example, the encoding "{text ff0080}" specifies a Red value of "ff", a Green value of "00", and a Blue value of "80". To revert the foreground text color back to the default, use a color value of "010101". NOTE! For additional details on specifying colors via the HTML syntax, please refer to one of the many websites that provide this information, such as http://www.w3schools.com/Html/html_colors.asp.
{back xxxxxx}	This sequence changes the background text color to the color given by "xxxxxx". The color format uses standard HTML syntax, as described for "{text}" above. To disable use of a background color and display text transparently, use a color value of "010101".
{align position}	From this point forward in the text, each new line of text will be aligned based on the given "position". The position must be one of the following values: "left" for left-aligned text, "right" for right-aligned text, or "center" for centered text.
{vert value}	Vertical spacing is immediately inserted into the output, with "value" indicating the number of pixels worth of gap to insert.
{horz value}	Horizontal spacing is immediately inserted into the output, with "value" indicating the number of pixels worth of gap to insert.
{offset value}	From this point forward, every new line of text has "value" pixels of horizontal spacing inserted at the start of the line. The offset persists, allowing large blocks of text to be inset from other text. You can turn off the offset by applying an equivalent negative offset. Repeated uses of "offset" are cumulative.
{indent value}	From this point forward, every new paragraph of text gains an indentation behavior dictated by "value". If "value" is positive, normal indentation logic is used and the first line of each paragraph is indented that number of pixels. If "value" is negative, a hanging indent is applied, with the second and subsequent lines being indented the specified number of pixels (as an absolute value). Repeated uses of "indent" are cumulative, except for a value of zero, which turns off any active indent behavior.

{bmp name} Inserts into the output the bitmap image with the file name of "name.bmp" that resides in the data file directory for the game system. This bitmap is assumed to be transparent, with the pixel at (0,0) indicating the transparent color. For example, the code "{bmp foo}" would insert the bitmap file named "foo.bmp" that is found in the game system directory. **NOTE!** If the bitmap is not found or the output doesn't support encoding, the "name" is inserted as text. So if the filename is "[R].bmp" and referenced as "{bmp [R]}", the text inserted would be "[R]". **NOTE!** By default, bitmaps are never scaled when output, but individual styles can explicitly enable scaling of bitmaps inserted via encoded text. If scaling is enabled, the bitmaps may not look ideal due to the scaling.

{bmpscale factor name} As {bmp name}, above, except the bitmap is scaled by a factor of "factor" (from 2-9) before being output. For example, if you create a bitmap "somebitmap.bmp" with a size of 400x400 and then output it using {bmpscale 4 somebitmap}, it will be drawn with a size of 100x100. This can be useful for drawing bitmaps at large scales, then shrinking them down for use on high-dpi mediums (such as printed pages). **NOTE!** This is only recommended for use on output sheets. Due to the windows bitmap resizing algorithm, results of this will likely be poor if used on the screen.

{meta behavior} The behavior specifies a new rendering behavior that will persist until it is again changed. The specified behavior must be one of the following values:

- ***bmpfull:*** Bitmaps embedded in the encoded text are vertically centered within the full height of the text, including the region of any descender.
- ***bmpbaseline:*** Bitmaps embedded in the encoded text are vertically centered between the baseline of the text and the top edge of the text. This is the default behavior used by encoded text.
- ***revert:*** All behavior changes applied via "meta" special codes are reverted to their default state.

{macro name} Looks for the macro with the unique id given by name and processes the contents of that macro as if it were found in the text instead. This means that macros MAY include encoded text that will be properly processed when the macro is evaluated.

NOTE! Macros MAY be nested within each other, but only to a limited extent, so use nesting sparingly. If a macro references another macro within it, the nested macro will be correctly processed, up to a small number of nesting levels. **NOTE!** If an undefined macro is referenced, a message of the form "[Undefined macro: name]" is inserted into the resulting text where the macro would be.

{ref name} Designates the start of a reference region (or "hot zone") that the user can click within to obtain additional information. The name specifies the predefined reference to show when the user clicks in the zone. The zone spans all text from this point forward until the zone is terminated. Terminating a reference zone is accomplished by leaving the name blank. The following is an example of using a reference: "before {ref foo}hot zone{ref} after".

NOTE! If an undefined reference is used, a message of the form "[Undefined reference: name]" is displayed when the reference hot zone is clicked within by the user.

{url name} Designates the start of a hot zone corresponding to an internet URL. The name specifies the actual URL to which the user will be sent when the zone is clicked within (e.g. "http://www.wolflair.com"). The zone spans all text from this point forward until the zone is terminated. Terminating a URL zone is accomplished by leaving the name blank. The following is an example of using a URL zone: "before {url http://www.wolflair.com}hot zone{url} after".

NOTE! When clicked, Windows is instructed to launch the default web browser that is currently configured by the user and to load the specified URL into that browser. If no browser is properly configured, an error will be reported. If the URL is invalid, the web browser will report that to the user.

\n Identical to "{br}". This is a programming convenience for automated conversion programs written in C/C++.

IMPORTANT! Not everything within HL makes use of encoded text, so be sure to check the documentation first. If you use encoded text in a place where it's not supported, you will see your embedded special codes within the text displayed.

NOTE! Encoded text sequences ignore case, so "{BR}" is identical to "{br}". Exceptions to this are macro and reference names, which are case-sensitive. Similarly, URLs are case-sensitive if the website being addressed uses case-sensitive URLs.

NOTE! If the text within an encoding does not correctly match one of the codes defined above, the text is left unchanged, except as noted above.

THE PHYSICAL FILES

There is an assortment of different file types involved in a complete set of data files. These files must reside in specific locations for HL to properly find and use them. In addition, there are critical naming conventions that the data files must subscribe to.

DATA FILE TYPES

HL utilizes a variety of different file types for defining all the particulars of a given game system. Each of the various file types is summarized below, with their specific contents being specified in the Kit Reference section:

- **Definition File:** The definition file provides the basic foundation for a particular game system, defining the fundamental characteristics that are shared across all aspects of that game. Each game system has exactly one definition file. Definition files are only utilized when creating data files for a new game system from scratch. See the Kit Reference section for details.
- **Structural File:** Structural files enable you to define the underlying structure of the game system, creating the framework on structural which all of the visual pieces and game mechanics elements will be defined. You can have any number of structural files, allowing you carve up the logic across multiple smaller and more manageable files. See the Kit Reference section for details.
- **Data File:** Data files are where all of the user-manipulated elements are specified for a game system. This includes both the visual elements (e.g. panels and forms) and the game system elements (e.g. classes, spells, skills, and equipment). Data files build upon the material in the structural files. See the Kit Reference section for details.
- **Configuration File:** When package files are used, the configuration file contains basic details about the game system, allowing HL to display appropriate entries for the game system for user selection. If packages are not used, then the configuration details always reside within the definition file and no configuration file should be present.
- **Package File:** Package files are pre-built collections of data files that are distributed by Lone Wolf Development. Packages provide a convenient way to distribute material to users, as well as offering security-restricted access. A package file that your license is not authorized to access will simply be ignored by HL when loading files. If your license grants access to a package, the package is equivalent to having all of the data files present within the game system data directory.

IMPORTANT! Although there is a specific file type of "data file", the general term "data files" is often used to collectively reference ALL of the various files which are created for a particular game system. Therefore, a reference to the "d20 System data files" would be referring to any definition file, structural files, data files, package files, and/or configuration file for the d20 System game system. When a reference is exclusively referring to a data file for a game system, it will either be spelled out within the Kit documentation or be clear from the context of the reference.

FILE LOCATIONS AND NAMING CONVENTIONS

Beneath the directory where you installed HL is a directory named "data". If you installed HL to the default location, this directory will be located at "C:\HeroLab\data". All HL data files reside in sub-directories beneath the "data" directory, with all of the data files for each game system being grouped together within a sub-directory named appropriately for that game system. For example, all of the data files for the d20 System will be found in the "d20" sub-directory, or "C:\HeroLab\data\d20" if you installed HL to the default location.

Within a game system directory, there will be an assortment of files with a variety of names. Here is a summary of the various files that will be found and their purpose.

config.cfg	Configuration file that provides details when package files are used
*.pkg	Package files
definition.def	Definition file that is compiled before anything else
*.1st	Structural files that are compiled first
*.core	Structural files compiled after "*.1st" files
*.str	Structural files compiled after "*.pri" files
*.aug	Structural files compiled after "*.pri" files
*.dat	Data files that are compiled after structural files
logo.bmp	Bitmap file containing game system logo
*.bmp	Image files used for buttons, textures, etc.
*.hlz	Compiled files used by HL at run-time
faq.htm	HTML file containing the FAQ for the data files that is generated when the data files are compiled
manual.htm	Manual with specifics on using the data files for the game system (file name may be different)
timing.xml	Timing dependency and error report generated whenever the data files are compiled
unused.xml	Report of unused resources and styles that is generated when the data files are compiled

FILE LOADING ORDER

As you may have spotted within the previous section, the Kit supports four different types of "structural" file. The information that can be placed in these files is identical, so this begs the question of why there are four types. The reason is that HL loads files in a specific sequence and all the material in one set of files must build on the previous set.

You can think of it as layers, where each new layer relies upon the layers beneath to be correct. During loading, each new layer of data files require that certain information it references already be in place

and verified. For example, tag groups that are used must be defined prior their use, else an error is reported.

In order to keep your data files more manageable, you'll likely want to keep them small and focused. Having a few huge files is perfectly acceptable, but we recommend against that approach and have structured the example data files accordingly. However, maintaining a variety of smaller structural files means that you'll be carving the information up across different files, and that can result in references across files that aren't defined.

Using tag groups as an example again, you might define a tag group in one file and then reference it in another. The problem is that you need to make sure that the file in which the tag is defined is always loaded before the file in which it is referenced. The only way to do this reliably is for HL to support different structural file extensions and load those files in a specific sequence. This way, you can put the tag groups in one file that you know will be loaded first and the references in other files that you know will be loaded afterward.

You can have lots of different structural pieces in a complex set of data files. And there are a variety of requirements regarding what information must be defined before it is used. Consequently, in order to handle complex situations, there are four different structural files. Putting all together with the other data file types, there are a total of six layers of files and they are loaded in the sequence specified below.

1. Definition file (definition.def)
2. Structure files with the ".1st" file extension
3. Structure files with the ".core" file extension
4. Structure files with the ".str" file extension
5. Structure files with the ".aug" file extension
6. Data files with the ".dat" file extension

DATA MANIPULATION BASICS

In order to handle the complexities of every game system, the HL engine is a very sophisticated tool. However, we've invested significant time and effort to keep things as simple and streamlined as possible. The net result is that the Kit makes extensive use of a few powerful mechanisms that give you complete control over both how data is manipulated and when it is manipulated.

Given the volume of information involved in many game systems, data manipulation entails the proper sequencing of tasks. The first key mechanism is the evaluation cycle that governs when the data is manipulated. The second key mechanism is a fast and flexible classification system, called tag expressions, which uses tags to identify which objects are to be manipulated and which manipulations should be applied to them. The final key mechanism is the scripting language that allows the actual manipulation of the data.

EVALUATION CYCLE BASICS

In order to ensure that all data manipulation operations are applied in a correct and consistent order, the HL engine enforces a strict sequence of evaluation that you completely control as data file author. This sequence is triggered repeatedly as the user makes changes to characters, and it is referred to as the "evaluation cycle".

Whenever the user takes any action, HL automatically re-evaluates all facets of the portfolio that are impacted by the change. This updates all dependencies on the user's changes. For example, if the user modifies an ability score in the d20 System, all linked skills and dependent weapons are updated to reflect the impact of the change. To safeguard against lag time in the user-interface when the user takes multiple actions in rapid succession (e.g. clicking the '+' button to increment a skill rating a dozen times), HL waits until the user pauses for a moment before it initiates a new evaluation cycle on the portfolio.

Everything that is performed by the Hero Lab engine is done in a specific sequence, and the majority of actions occur during the evaluation cycle. Each of these individual actions is referred to as a "task", and the complete set of tasks comprising the evaluation cycle is referred to as the "task list". For virtually every task, the data file author controls the evaluation sequence by designating when each task should be processed. There are two criteria used to determine the scheduling of a task: phase and priority.

For each game system, the data file author defines a set of phases that dictate the general sequence in which evaluation is performed. Each phase typically corresponds to a logical step in the overall evaluation cycle, such as "initialization", "before level-based calculations", "after attribute modifiers", etc. All phases are ordered, thereby dictating the sequence in which the phases are processed during the evaluation cycle.

Every task is assigned a phase during which it will be evaluated. All tasks are also assigned a priority, which controls the order in which tasks within the same phase are processed. If two or more tasks have the exact same phase and priority, then the engine uses a number of rules to order them. If the two tasks are still scheduled for the same time, the engine is free to schedule them in whatever sequence it finds convenient, and this order **may change** from one evaluation pass to the next. Consequently, assigning the correct phase and priority is often critical to ensure that modifications are applied before or after subsequent tests are made that rely on those modifications.

The Kit provides an assortment of pre-defined phases that should serve as an excellent starting point for just about any game system. You are free to change or delete these phases, as well as add new phases to suit your needs. However, the set provided will typically work well for most game systems we've encountered.

Report of unused resources and styles that is generated when the data files are compiled when the evaluation cycle begins, it continues until completion. This is usually transparent to the user, but it can become noticeable on older (i.e. slow) computers when the data files are highly complex. Therefore, it's best to utilize tag expressions whenever possible to limit the number of objects that must be processed during evaluation. Similarly, it's typically faster to use tags instead of scripts when possible, because they are significantly faster to execute.

HOW TAGS WORK

Virtually everything within the Kit leverages tags in some way. Tags represent a generic mechanism for assigning arbitrary attributes to a thing, pick, or container. The only restrictions regarding tag assignments are those that are defined for a given tag group (e.g. minimums, maximums, duplicates, etc.). This allows the data file author to design the set of tags and tag groups specifically for the behaviors of a particular game system. With this simplicity comes a great deal of power and flexibility. Unfortunately, the lack of a regimented structure also introduces some potential confusion when

getting started. The rest of this section will hopefully point out some useful pointers to get you underway.

We'll start with an in-depth review of the tags model. To use a real-world analogy, consider a database of shirts. Every shirt has a color, a sleeve length, a size, a style (e.g. t-shirt, polo shirt), and various other characteristics. Each of these would translate to a tag that is associated with a given shirt. There would be one group of tags for the color, with a separate tag for each color. Another group would reflect the size, with separate tags for each size. So all you need to do is associate the proper tags with the shirt to describe it. Collectively, all the tags assigned to the shirt comprise its description.

So what if you are presented with a shirt that is striped in red and white? Do you classify the shirt as red or white? Your first thought might be to define a new group of tags for the design of the shirt, with possible tags including stripes, polka dots, plaids, etc. But that still doesn't address the issue of color. The goal is to easily search for shirts in your database, so the best choice is to declare the shirt as both red and white, assigning it both tags. This way, if the user is looking for a shirt, the user can decide whether he wants a shirt that is only red, a combination of red and another color, or specifically red and white. Using tags, you can easily make this distinction.

Let's take this a step further. What if you have some shirts with two-colors of stripes and other shirts with three (or even four) different stripes? One option would be to have different design tags for each number of stripes. But what if a user just wanted to find all the red striped shirts, without worrying about how many stripes? The tags method provides two ways of easily handling this situation. The first technique is to assign the same stripes tag to the shirt multiple times to indicate the number of stripes on the shirt. This allows the user to query striped shirts in general (at least one stripes tag assigned to the shirt) and shirts with varying numbers of stripes (the shirt must have X number of stripes tags assigned).

The second technique is to define a separate tag for each number of stripes, but do it so that wildcards can be utilized. For example, defining the tags "stripes2", "stripes3", and "stripes4" would make it possible to assign the proper tag corresponding to the number of stripes on the shirt. The user could query striped shirts in general by using a wildcard to identify any tag that starts with "stripes". Querying for an exact stripe count would simply look for shirts with the specific stripes tag.

All of the scenarios described above will occur regularly when designing data files for role-playing games. They will be useful for validation rules, condition tests, and managing when and how to apply adjustments to picks and containers. But the key detail is that the tags mechanism provides a single, generic mechanism that can be creatively used to handle virtually all design situations – both efficiently and without a high level of complexity for the author. The trick is in picking the right set of tags to make your life easier as a data file author.

To help you with that last consideration, the starting data files included within the Kit provide an assortment of tags and tag groups to handle many of the most common situations. These tags are all utilized within the starting data files, so you can see concrete examples of how they can be put to use.

TAG EXPRESSION BASICS

Tags form a fundamental building block upon which much of HL is constructed, and tag expressions are where they become of critical importance. Since the vast majority of objects you'll be managing are things and picks, there must be a way to identify the proper subset of these objects that apply to a particular situation. For example, attributes, skills, and weapons are used in completely different ways, so you want to keep them separate from each other – yet they are all things (or picks). The solution is to assign tags to each object and then use a tag expression (or tagexpr for short) to identify the subset of objects that apply in a given situation. A major (separate) section of the documentation is dedicated to the subject of tag expressions, but a brief overview is valuable at this point.

A tag expression is essentially a filter that gets applied to all objects of particular type (e.g. things or picks) and selects only the ones that meet the specified criteria. Tag expressions are Boolean expressions, which mean they evaluate to a simple "true" or "false" result. They examine all of the assigned tags and determine whether those tags satisfy the expression or not. Separate criteria can be combined, allowing you to require that multiple criteria all be met, one of a set of criteria be met, certain criteria be excluded, or some combination thereof. For example, a tag expression could test whether a thing has the tag "Elven" from the "Language" group. Or a more complex tag expression could test whether a thing has the "Language.Elven" tag and also has either the "Race.Elf" or "Race.HalfElf" tag.

Since tag expressions can utilize full Boolean logic (i.e. "and", "or", "xor", "not") and can even extract and test numeric values from tags, tag expressions can model extremely complex conditions without difficulty. The bottom line with tag expressions is that they provide a powerful and flexible method for quickly determining whether to include or exclude an object, and they are based exclusively on the set of tags assigned to that object. As such, they are used extensively throughout HL.

SCRIPTING BASICS

HL makes extensive use of scripts to allow the data file author substantial freedom and flexibility. In fact, scripting is such a fundamental and diverse topic that huge sections of the documentation are dedicated to various facets of writing scripts. This section merely provides a brief overview.

The scripting language syntax within the Kit is relatively simple. You can declare variables, assign values, perform simple conditional tests, and utilize a number of built-in intrinsic functions for various purposes. There is also a syntax that allows access to all of the objects within a given actor, such as the various picks and containers, plus the field values and tags that may be assigned to them. The language syntax itself is somewhat similar to the age-old Basic language. Using scripts, an author can pretty much do whatever is necessary to properly model the requirements of a given game system.

To make the writing of scripts easier, the Kit supports re-usable procedures. A procedure is nothing more than a mini-script that can be called from multiple places. The data files for each game system make extensive use of procedures so that many scripts can be reduced to simply calling one or two procedures to do all the work.

The starting data files provided by the Kit include numerous scripts and procedures that you can use, adapt, and extend according to the needs of the game system you're implementing.

THING-BASED SCRIPTS

The most common type of script that you will find yourself writing is the "eval script". Of all the different types of scripts, most are

triggered in response to a specific action. However, an eval script is scheduled to occur at a specific phase and priority during the evaluation cycle, hence the name.

Every eval script is associated with a thing. This means that every pick derived from a given thing inherits all of the eval scripts for that thing.

Since an eval script is performed during evaluation processing, a separate task is always created for each eval script, which is then scheduled within the overall task list. If a thing is added to a container multiple times, each eval script must be processed separately for every pick. Consequently, separate eval script tasks are created and scheduled for every pick.

Eval scripts are the most commonly used script due to the ability to schedule them. So many facets of a complex game system are inter-dependent. Dependent calculations must be performed in a carefully ordered sequence to ensure that all of the game mechanics are accurately implemented within the data files. Eval scripts provide the means for this scheduling.

THING-BASED RULES

Sometimes, you don't need to apply a modification to anything and instead need to verify whether a required condition is satisfied. For example, the "Knowledge" skill might require that the user specify the domain to which the skill applies. This entails checking that the user has entered a suitable value.

To handle situations like this, the Kit provides "eval rules" as a companion to eval scripts, and you will likely find yourself writing a number of these as well. Just like eval scripts, eval rules are scheduled as tasks and performed with specific timing during the evaluation process. Unlike eval scripts, the purpose of an eval rule is to verify a condition, so an eval rule must always determine whether its condition is satisfied or not. If the rule is not satisfied, the corresponding pick is flagged as being invalid and an appropriate message is displayed to the user within the validation report.

COMPONENT-BASED SCRIPTS AND RULES

There are many situations where you'll want the same script or rule to be applied to all things that derive from a particular component. For example, every piece of gear the character is wielding should be verified to not be stored in a backpack or some other container. The Kit makes it easy to handle situations like this by allowing you to actually define the script or rule directly on the component.

Whenever a script or rule is defined on a component, that script/rule is treated as if it were individually assigned to every thing that derives from the component. This means that a new task is automatically created and scheduled for the script/rule on every pick that is added to the character.

It is reasonably common to have both component scripts/rules **and** individual script/rules associated with a particular thing.

INFORMATION ACCESS

The goal of writing a script is to retrieve and/or manipulate one or more facets of either an actor or a visual element. Consequently, your primary focus, as an author, will be on how to access information. The Kit uses an hierarchical structure for managing all the different objects. This section of the documentation outlines the basics of how to navigate the hierarchy and access the information you need.

SCRIPT CONTEXTS AND TRANSITIONS

Every script begins with an initial context. The context depends entirely upon the script, but the context always refers to a specific object maintained by HL, such as a pick, an actor, or a layout. For example, an Eval Script begins with an initial context of the pick that the script will be executed upon. Similarly, a Position script for a layout begins with an initial context of the layout for which the contents need to be positioned on the screen.

When you are writing a script, you will often need to retrieve information from or apply changes to objects other than the initial context your script begins with. This is accomplished by performing a context transition to the context of a connected object within the hierarchy, thereby establishing the new object as the new context. For example, you can transition from a pick context to the context of the container that contains the pick. Or you can transition from a pick context to the context of one of its fields.

Using context transitions, you can "travel" through the hierarchy of objects (within limits), seeking out the specific objects you need to manipulate via your scripts. The exact list of what context transitions are possible is entirely dependent on the script and the current context. A complete list of contexts and specific transitions available is outlined in the Kit Reference documentation.

IMPORTANT! The initial script context resets for each separate identifier used within a script - it does not persist. This makes it easy to have a single statement access multiple different contexts by transitioning in different directions from the same initial starting point.

NOTE! It is possible to transition to an invalid (i.e. non-existent) context. For example, a script might try to transition from a container to a pick that doesn't exist within that container. If that occurs and the non-existent context is then accessed (e.g. getting the name of the non-existent pick), a run-time error will be reported and the access will fail.

SCRIPT TARGETS

Once you have established the final context to operate upon, you need to identify the specific facet of the context to access or manipulate. This is referred to as a target reference, or sometimes simply target. For example, if you have a pick context, you must specify whether you are interested in its name, its tags, or something else.

The syntax for accessing a given target aspect will vary, depending on what the target is and what information is needed to uniquely identify the desired target. All of the different target references are defined in Kit Reference documentation.

TARGET IDENTIFIERS

In order to access or manipulate a component, your script must specify both the intended context and the target. Scripts use an open- ended dot notation to convey this information and uniquely designate a target identifier. A target identifier represents the combination of context and target reference, which should yield a specific piece of information to operate upon. You can use valid target identifiers anywhere within scripts that you would use a simple variable, allowing you to retrieve the current value of a target identifier or set that value.

The target identifier consists of a sequence of zero or more context references, each separated by a period ('.'), and finally followed by a

single target reference. In sequence, each context reference must indicate a valid context transition from the previous context. The target reference must be valid for the final context established. This syntax for specifying a target identifier is referred to as the dot notation.

TYPES OF SCRIPTS

The Kit supports more than 40 different types of scripts that make it possible for you to control virtually every facet of how the data files behave. Some scripts control visual facets of how information is presented, while others control the actual manipulation of the character. In general, all of these types of scripts can be distilled into a handful of different categories.

The list below presents a summary of the various categories of scripts, while the complete list of all script types will be found in the Kit Reference documentation.

- **Visual Positioning:** Scripts that manage the size and positioning of visual elements within panels and sheets
- **Synthesis & Presentation:** Scripts that synthesize information for display to the user in some fashion, including labels, descriptions, mouse-over information, and stat blocks
- **Pick Manipulation:** Scripts that manipulate the contents of picks during the evaluation cycle
- **Field Manipulation:** Scripts that manipulate the contents of fields for both display and constraint
- **Validation:** Scripts that apply validation tests to objects with integrated reporting of errors
- **Creation/Deletion:** Scripts that perform appropriate setup and cleanup of specialized objects
- **Transaction:** Scripts associated with the buying and selling of equipment
- **Mode Transition:** Scripts associated with the transition into and out of advancement mode
- **Trigger:** Scripts that are invoked in direct response to user actions, such as merging and splitting stackable gear, controlling combat and turns, etc.
- **Fixup:** Scripts used to accommodate changes between data file releases and potential loading errors of portfolios

MANIPULATION OF VISUAL ELEMENTS

In addition to getting the underlying game mechanics correct, you'll also need to effectively present the information to the user and allow him to create a character. The overall hierarchy of visual elements has already been described. This section delves into the process of successfully manipulating those visual elements to achieve an interface that works smoothly.

SCREEN VS. PRINT OUTPUT

The Kit makes a critical distinction between output to the screen and output to the printer. Similarly, there are completely separate portals for the purpose of screen output versus for printed output. Layouts and templates are classified for screen versus print output based on their contents. Panels, forms, and sheets are only allowed to contain layouts that are designed for screen or print output, as appropriate to their nature. Sheets are intended for print output, while the other two are intended for screen output. This begs the question of why the distinction is made.

The obvious distinction is that portals used on the screen will typically involve some means for the user to modify content, while portals for printed output are for display only. However, there are a number of on-screen portals that are solely for display purposes (i.e. labels, images, and fixed tables). In fact, those on-screen portals are the exact ones that have equivalent versions for printed output, so why couldn't those portals be the only ones allowed for output use? Because there is more going on that may not be immediately obvious.

The key reason for the separation is that portals for printed output have some critically different needs from on-screen portals. By keeping the portals distinct from each other, those different needs and behaviors can't be inadvertently confused by authors. And by percolating that distinction up through the entire visual hierarchy, everything is kept clear for authors.

So what are some of these important distinctions? A quick sampling is provided below.

- Portals for output have different style needs from on-screen portals. Whether it be the use of color, special borders, or something, the visual for printed output are significantly different from on-screen display.
- Different portals need to behave differently within printed output. For example, the titles drawn above major sections of printouts don't work the same and have different needs from the titles drawn above sections on-screen.
- For printed output, it is commonplace to need items of varying height within tables. On the screen, it's more valuable to use fixed height items with mouse-over regions so that everything is kept compact and easily scrolled through for long lists.
- The scaling logic used for images can be different when displaying on a comparatively low-resolution screen versus on a high-resolution printer.

The list goes on, but this should give you a good idea of why the distinction between printed and on-screen output is important.

HOW VISUAL ELEMENTS BEHAVE

Visual elements share a number of behaviors that are common across most, if not all, different visual types. These common behaviors are discussed in the topics below.

RECTANGULAR REGION

Every visual element represents a rectangular region, whether it is on the screen or on the printed page. As such, each element possesses both dimensions and a position. The dimensions are a simple width and height. The position is given as the location of the top left corner of the visual element within its containing visual element.

The position is always relative to the containing element and consists of two values. It represents a number of pixels along the X axis and a number of pixels along the Y axis. Values along the X axis increase as you move from the left to the right. Values along the Y axis increase as you move from the top to the bottom. By convention, a position always lists the adjustment along the X axis first, so a position of (7,42) indicates 7 pixels to the right along the X axis and 42 pixels down along the Y axis.

As an example, consider a portal that is placed at position of (10,0) within a template. Then the template is placed at a position of (5,0) within a layout. The layout is placed at a position of (2,0) within a panel. Putting it all together yields an effective position of (17,0) for the portal within the panel.

When modifying the rectangular region for a visual element, there are two different ways to accomplish it, and each yields different results. The first method is to set the width and/or height for the element. This keeps the top left position anchored in place and merely moves the bottom right position. The alternative is to directly modify the right and/or bottom edges of the rectangle, which implicitly changes the dimensions assigned to the visual element.

MARGINS

All visual elements that can contain other elements possess margins. The margin is a gap that is maintained between the outer edges of the visual element and the interior region where child elements are positioned. A vertical margin is maintained separately from any horizontal margin, and both default to zero.

For the purposes of placing child elements within a visual container, the zero position is equal to any margin value. Normally, when placing a portal at a position of 5 within a template, the portal appears at the position of 5. However, if the template has a margin of 3, that gets added to the specified position. This yields an effective position of 8 for the portal within the template.

Margins accumulate at every level within the hierarchy. Let's return to the example used in the previous topic about a portal within a template within a layout within a panel. If each of the visual contains has a margin of 3, then each successive child element is shifted 3 pixels further to adjust for the margin. This yields an effective position of 26 for the portal within the panel.

VISIBILITY

Every visual element has a visibility state, which can either be visible or hidden. By default, all visual elements start out as visible. However, you can set the visibility dynamically within any script that positions a portal. If a visual element is designated as hidden (i.e. not visible), then all visual elements within that element are hidden as well. This continues down through the hierarchy. Consequently, setting a layout to non-visible implicitly hides all templates and portals within the layout, plus it also hides all portals within the now- hidden templates.

There will be many places where you will want to control the visibility of visual elements.

THE POSITIONING SEQUENCE

THE POSITION SCRIPT

Every visual container possesses a "position" script, which means the script is possessed by every template, layout, panel, and sheet. The Position script serves one specific purpose - to properly size, position, and configure all of the child elements within the visual container. Not every visual container needs to define a position script, since the HL engine performs some actions automatically, but most visual containers will perform at least some operations within their Position, and some will perform extensive operations to properly setup the visual elements within them.

RECURSIVE DESCENT THROUGH HIERARCHY

The HL engine utilizes a consistent process for positioning all of the visual elements. Each top-level element (i.e. scene) is handled independently. Within the context of each scene, a recursive descent is performed upon all of the visual elements it contains, invoking the Position script within each visual element during the descent. How this works is detailed in the sections below.

Prior to doing the recursive positioning, all visual elements within the scene are properly reset. This entails setting their position to (0, 0), performing default sizing, and initializing their default state. In general, only portals possess default sizing and state.

POSITIONING SCENES (PANELS, FORMS, AND SHEETS)

All positioning starts with a top-level visual element: either a panel, a form, or a sheet. Within the scene, the Position script is invoked. After the script returns, any child layout that has not yet been rendered is now rendered.

Within the Position script, the panel or sheet may need to force a layout to properly calculate its size before another layout can be positioned relative to the first one. When this occurs, the panel or sheet can explicitly tell the layout to render itself by using the "render" target reference on that layout. Rendering a layout invokes the Position script for that layout immediately. Since each layout is automatically rendered at the end, triggering the render from the Position script is only necessary if the rendered results are needed within the script.

Rendering a layout does not place anything on the screen or page. All it does is trigger appropriate sizing, positioning, and state configuration. It is perfectly valid to render a layout multiple times. In fact, there will be situations where you will want to do this to optimally position information. You can render a layout, find out about size of the layout, force a change to the layout, then render the layout again.

NOTE! Rendering a layout multiple times is computationally expensive. Consequently, you should limit re-rendering to only be used when it is truly needed.

POSITIONING LAYOUTS

When a layout is positioned, its Position script is invoked. Once the script returns, any child templates that have not yet been rendered are now rendered.

Just like with panels and sheets, the layout may need to force a child template to calculate its size before a separate visual element can be positioned relative to it. You can explicitly trigger a template to be rendered by invoking the "render" target reference on the template. As above, rendering a template immediately invokes the Position script of that template. This is not always necessary, since the layout will automatically render all templates after the Position script completes.

In the same way that layout rendering does not actually output anything, template rendering simply determines the position, size, and state of the template and its contents. Nothing is actually output, so it is valid to render a template multiple times, and there may be situations where you need to do that.

NOTE! Rendering a template multiple times is computationally expensive. Consequently, you should limit re-rendering to only be used when it is truly needed.

Positioning Templates and Portals

When a template is positioned, its Position script is invoked. This orchestrates the sizing and positioning of all portals within the template. Unlike the above visual elements, when the Position script returns for a template, there is nothing more to do for that template and/or its contents.

Positioning Portals Within Templates

Due to the way the Kit is designed, the single most common thing you will find yourself doing with visual elements is positioning portals within templates. To better streamline development, the Kit has a number of special optimizations that can be used to efficiently position portals within templates. The topics below discuss some of these shortcuts.

IMPORTANT! The various target references described below are **only** usable on portals within **templates**. If you attempt to use them on portals within layouts (e.g. tables and choosers), they will not work.

Automatic Sizing

The HL engine includes logic that will intelligently determine the dimensions of portals. For example, a label portal will have its size calculated to match the text it contains, an image portal will be sized to match the image it contains, etc. If an appropriate default size cannot be determined, then something safe is used instead.

This mechanism is referred to as automatic sizing and is accessible at any time via scripts. In addition, you can trigger the default sizing logic to be applied to only a single dimension instead of both width and height. As an example, if you change the style of a portal and thereby change the font size used, you will need to re-size the portal based on the new style. Using the automatic sizing logic is a quick and easy way to handle it.

Default Sizing

At the beginning of the visual positioning logic, every portal is assigned an appropriate default size. This is achieved by determining the automatic sizing characteristics for the portal, as described above. As a result of this default sizing, you often don't have to worry about setting the dimensions of portals and can simply focus on positioning them.

Size of Contents

For text portals especially, but sometimes for other portals, you will want to explicitly retrieve facets of their contents. There are three different types of sizing information that can be retrieved. The first is the "text width", which retrieves the width necessary to contain the full text on a single line. The second is the "font height", which retrieves the height of a single line of text, based on the font characteristics established by the current style assigned to the portal. The third is the "text height", which calculates the number of lines of text needed to display all of the text contents and the full vertical extent of that text.

The "alignedge" Target Reference

When positioning portals, a few will be positioned based on the edge of the template, with the rest being positioned relative to the other portals. When you need to position a portal relative to an edge of the template, you can use the "alignedge" target reference. You identify the portal, the edge (left, top, right, bottom), and the offset to use. The offset makes it easy to leave gaps from the edge if you need them.

The "alignrel" Target Reference

Once you've positioned the initial portals relative to the edges of the template, it's now time to start positioning portals relative to each other. To simplify this process, the "alignrel" target reference is provided. You start with the portal to be positioned and then specify the portal to position relative to. You then add the positioning relationship to be used and any offset to be inserted. The positioning relationship identifies both the edge of the portal being placed and the edge of the reference portal, and any combination is allowed. For example, a "left to right" relationship means that the left edge of the portal being placed is positioned relative to the right edge of the reference portal. If an offset of 10 is given, then the portal would be placed 10 pixels to the right of the reference portal.

The "centerhorz" and "centervert" Target References

You will quite often need to center portals along one axis within the template. To accomplish this, all you need to do is invoke the "centerhorz" or "centervert" target reference on the appropriate portals. The HL engine will calculate the relative dimensions and properly position the portal to be centered along the specified axis.

The "centeron" Target Reference

There are times when you need to center multiple portals along a common central axis. In a situation like this, you must first position one portal where it needs to be. After that, you can position the remaining portals relative to the original portal by using the "centeron" target reference. The HL engine will calculate the center line for both portals and position the new portal appropriately.

The "centerpoint" Target Reference

When setting up columnar tables, you will sometimes want to center data within columns. There are two ways of handling this. First, you can configure each portal to center its contents within itself and then set the width of each portal to match the column width being used. The drawback to this is that the default sizing logic will set each portal to an appropriate width based on its contents, and that means that you'll need to reset the width of every portal and position it properly.

The alternative is to calculate the center position of the column. Once you have that, you can simply use the "centerpoint" target reference to position each portal. The HL engine will determine the width/height of the portal and position it so that it is centered on the point specified.

Dynamically Changing Styles

Every portal is assigned a style, and that style dictates a variety of visual characteristics about the portal. They can include the font size, font style, text color, background color, border, bitmaps used, and an assortment of other attributes. In a nutshell, the style is what

gives a portal its visual look and behavior. However, the style is not fixed - it can be changed.

There will be many situations where you'll want to highlight a particular condition to the user. If you think of the style as fixed, then you're likely to conclude that the best way to switch from a normal looking label to one that is in red text and a large bold font is to have two different portals. You can then hide whichever portal is not applicable based on the prevailing condition state. While this approach will definitely work, it entails extra effort.

The better approach is to simply change the style used on the portal. Since you'll need to define the alternate style to support a second portal, there is no extra work involved there. Then, within the Position script for the containing template, simply check the condition and, if appropriate, use the "setstyle" target reference on the portal.

The only caveat that you'll need to worry about with this approach is that you have to be aware of changing the font size and/or font style. If you change either of those characteristics with the new style, the default sizing of the portal will become invalid. Fortunately, all you need to do is then use the "autosize" target reference on the portal to trigger the appropriate resizing. Just make sure that you do this **before** you position the portal within the template.

KEYBOARD TAB ORDER

HL provides full keyboard support for navigating around the interface. This includes the ability to use the <Tab> key to move through the various visual elements via the keyboard.

TAB ORDER WITHIN TEMPLATES

Authors can easily control the order in which portals are moved between within a given template. The tab sequence for portals is simply the order in which the portals are defined within the template. If a portal is listed as the very first portal within the template, then the HL engine assumes that portal should be given the keyboard focus first. If the user presses the <Tab> key, then the next portal defined within the template becomes the new keyboard focus. This process continues until the last portal in the template is reached.

If a portal within the template cannot be given the keyboard focus, the focus goes to the next portal, continuing until a valid portal is found. Some portal types, such as labels, can never receive the keyboard focus. Other portals can be made non-visible or disabled via scripts, which render them unable to receive the keyboard focus.

TAB ORDER WITHIN TABLES

The tab order within a table gets a bit interesting. The following sequence outlines the logic used for tables.

1. If the table has a header template, then any portals within the header are first assigned the focus.
2. After the header, the table itself is considered as being a selectable portal via the keyboard, so it receives the focus next. This is only performed for tables that are scrollable, since given the focus to the table allows it to be scrolled via use of the arrow keys
3. After the table itself, control passes to the first item within the table. The template associated with the first item dictates the tab order for portals displaying information about the first item
4. Control passes through each item in the table, in order, with the template again dictating the tab order for that item.
5. If the table has an add option or footer at the bottom, control passes to it. The template associated with the add option or footer dictates the tab order for portals therein.

Once control passes from the last portal within with the table, it transfers to the layout that contains the table.

TAB ORDER WITHIN LAYOUTS

Within a layout, the tab order of the various visual elements is controlled by the "taborder" attribute that can be assigned by the author. Both portals and templates are mixed together for the purposes of determining tab order, and they are sequenced in increasing order based on the value assigned.

When a template is next in the tab order, HL will transfer control to the first portal within the template that can accept the focus. The focus will continue through to the last portal within the template. Once the last portal is reached, the focus will be transferred to the next template or portal specified by the layout.

When a portal is next in the tab order, HL will set the focus to that portal. If the portal contains other portals (e.g. a table), then the focus will continue to move through the contents of the portal. Once the final child portal is reached, the focus will be transferred to the next template or portal specified by the layout.

TAB ORDER WITHIN PANELS AND FORMS

Within a panel or form, the tab order of the contained layouts is dictated by the sequence in which the layouts are specified within the panel. When a panel/form is shown, the first layout is given the focus, and the appropriate rules for layouts are used to establish the initial focus. When the user moves through the last visual element within the layout, the next layout is given the focus, and the process continues. When control returns from the final layout within the panel/form, the focus is given to the first layout, and the entire cycle begins anew.

WORKING WITH TABLES

Table portals behave in significantly different ways from other types of portals. Since you will be using tables extensively in your data files, it's critical that you become familiar with the nuances of working with tables.

CONTROLLING TABLE CONTENTS

The purpose of tables is to display a collection of picks or things that are related in some way. For example, all the attributes for a character are shown in a table, all the skills are shown in a table, all the weapons and armor are broken up across multiple tables, etc. So the first order of business with a table is identifying the set of picks or things to be viewed.

Tables are generally associated with a specific component. This component establishes the first criteria for determining which objects are displayed in the table. If an object derives from the specified component, then it is valid for inclusion in the table. In many situations, this is all that's required.

However, there are times when additional filtering is required. In this case, you can specify an additional tag expression that is applied to every object that derives from the component. This is referred to as the "List" tag expression, since it identifies which objects are

shown within the list of objects for the table. Any object that satisfies the List tag expression are included in the table, while any object that does not is omitted.

Tables Use Templates

Tables make extensive use of templates. All of the objects shown within a table are viewed through a template. If the table is dynamic and allows items to be added, a separate template is used for displaying the items available for selection. If a table has a header or footer, then a template is used to display that information.

In fact, everything displayed as part of a table is accomplished using a template. The reason for this is simple. Using templates makes tables highly modular, allowing authors to readily customize the contents and re-use common templates for similar purposes.

How Templates Are Used

Every item shown within a table is either a thing or a pick. Every item that can be selected for addition to a table is a thing or a pick. Templates are an ideal vehicle for display within tables, since templates are always associated with a specific thing or pick. When used for showing a list of items, the same template is simply associated separately with each item in the list to properly render the contents of each item.

When rendering the table to the display, the HL engine will retrieve the list of objects that belong to the table. Then each of those objects is rendered, one at a time, into the table. For each object, the template defines the portals to be displayed and the object provides the appropriate fields from which to pull the information. Since the template is rendered separately for each item, it's perfectly reasonable to customize the contents of the template for each object. For example, certain portals can be hidden for some objects, while styles can be changes for certain portals on other objects.

The Show Template

Every table uses at least one template - the "show" template. This template is used to display each of the items within the table.

Reusing Table Templates In Multiple Places

Templates are not restricted to where they can be used. Consequently, you could choose to use the same template within a table and outside of that table. Or you could use the same template within multiple tables. Doing this is extremely useful. The problem is that, if you do share the template, you will probably need to make a few adjustments to how everything is handled within the template based on its context. The Kit makes this possible by providing the "intable" target reference that can be used on within the template's Position script. This target reference can be used to determine whether a template is within a table in general or to determine which specific table it is being used within.

Sequencing Through Sort Sets

Tables utilize sort sets to specify the sequence in which objects are displayed. There is no requirement that a sort set be given for a table. If omitted, then all of the items are simply listed in case-independent, alphabetical order.

One sort set is used to dictate the order in which the table contents are shown. However, a separate sort set can also be specified for dynamic tables. Dynamic tables provide the ability to select items to add to the table, and the available items are presented in a list. So the second sort set controls the order in which the items are shown within the list of available items for selection.

Adding Items to Tables

One of the most common mechanisms you'll be using is dynamic tables, as they allow the user to add new items to the character. Whether it is for skills, abilities, powers, spells, gear, or something else, a major part of character creation involves selecting an assortment of options from a list to customize the character. Dynamic tables are the mechanism the Kit uses to accomplish that task.

The Choose Table Vs. Show Table

Dynamic tables are actually two tables in one. First, there is the table in which the selected items are displayed, often called the "show table", and this works as outlined previously. However, there is a second table involved, which is the means through which available items are presented to the user for selection. This is often referred to as the "choose table".

The choose table behaves very similarly to the show table. It uses the same component for filtering the list of items shown. It also uses the List tag expression to filter the items shown. The reason for this is that any item that gets added to the table must fundamentally be shown within the table, so it stands to reason that the same requirements be applied.

The Candidate Tag Expression

However, the choose table also has a **separate** tag expression, which is called the "candidate" tag expression. The Candidate tag expression is distinct because there are often times when you'll want to be more restrictive in deciding which items can be added to the table. Depending on your preferences, the Candidate tag expression can either supersede or be **combined** with the List tag expression to yield the final list of items that are presented to the user for selection.

For example, consider a table of special abilities. Many abilities will be user-selectable, while a number of them will only be added to the character as the result of another choice, such as racial abilities being automatically added only when the corresponding race is chosen. The list of abilities possessed by the character and presented in the show table must include these racial abilities, while the list of abilities that the user can freely choose from must not. To handle this distinction, two separate tag expressions are required.

The Choose Template

In addition to the separate tag expression, the choose table also has a separate template. The list of information users want to view when choosing an item will often be different from the information they want when viewing items that have been added to the table. To accommodate this, a separate template is specified, but there is no reason that the same template cannot be used for both purposes. By decoupling templates from tables, you can readily re-use the same template in multiple places.

Choosing Things Vs Picks

Another important distinction between the choose table and the show table is the nature of the items presented for selection. Every item that has been added to the character is always a pick, which means that the show table always displays an assortment of picks. However, the choose table can contain either picks or things (never a combination). The norm will be things, such as presenting a list of

weapons that the user can select from. When things are chosen, a new pick is created and that pick is then added to the character.

Sometimes, though, you will want to display picks that have already been added to the character. For example, the Wizard class in the d20 System must memorize spells from the spellbook. When spells are added to the spellbook, things are chosen and picks are created. However, when spells are memorized, the list of available spells is pulled from the picks that have already been added to the spellbook.

PRESENTING THE ADD OPTION

For consistency, adding an item to a table is always managed through a special option at the bottom of the table. There are two ways to configure the add option. One method is quick and easy, while the other entails more work but provides greater flexibility.

The quick and easy method is to define an "AddItem" script as part of the table. This script specifies the text to be displayed for the add option at the bottom of the table. You can use encoded text to including color highlighting and other formatting, but you are limited to a single line of text. Since a script is involved, you can change the behavior based on the prevailing conditions, such as greying out the text if all available slots have been used up or turning it red to indicate too many slots have been used. If you use this technique, the Kit will automatically handle all of the mechanics of presenting the add option.

The alternative is to define a custom template and specify the object to be used with that template. As with everything else for tables, the display of the add option is controlled through a template, and every template must have an object from which is can pull its information. So you can create your own custom template that contains any portals you deem necessary. This technique is rarely necessary, but it does come in handy in some situations. For example, within the d20 System data files, all gear is ascribed a size rating, and each piece of gear must have its size specified when added to the character. So the d20 System data files use a custom template for the add option that includes a menu whereby the size can be specified.

CUSTOMIZING THE CHOOSE FORM

In addition to the choose table described above, the form presented for item selection contains other elements. The contents of the choose form can be customized appropriately for whatever is being added. Some of the key elements are discussed below.

TITLE BAR CONTENTS

Across the top of the choose form is a title. By default, this title contains something generic, along the lines of "Choose an Item from the List Below". While this may be sufficient, it doesn't really tailor the form to a clear purpose. It would be significantly better to even have a title like "Select a Weapon from the List Below". And in situations where the user is selecting items that are limited (e.g. choosing two starting special abilities), it would be even better if the title provided feedback about how many selections remain.

To make this sort of customization possible, dynamic tables possess a "TitleBar" script. The script allows you to construct a suitable title that conveys whatever information is going to be most useful for the user. For example, in Savage Worlds, each starting character is given 15 skill points that can be used to add new skills and improve them. So the Savage Worlds data files display the number of remaining skill points within the title above the choose table for skills. When skill points remain, the title is highlighted in yellow. When they are all used, the title turns grey. And if they are overspent, the title turns red. This type of customization is easy and makes the data files significantly friendlier to use.

DESCRIPTION REGION

To the right of the choose table, a large region is provided in which the detailed description of the currently selected item is shown. In some cases, you will find that the default width of this region is too narrow for the items being presented. When this happens, you can use the "descwidth" attribute on the table to change the width of the description region.

You can also control the contents of the description region if you want. This is achieved via the "Description" script for the table. By default, the description region will contain the name and description text for the currently selected item. However, you can customize this content to include more detailed information by defining a suitable Description script.

BUY/SELL TRANSACTION

Support some types of items involve a transaction for buying and selling the item. For example, weapons and gear will often have a cost associated with them, and characters will have some amount of currency. By integrating the process of buying and selling items directly into the adding and deleting of items, users are able to easily manage their expenses by allowing HL to track everything for them.

For items that entail buy and sell transactions, you can specify suitable templates to be used for each purpose as part of the table. When the choose form is displayed, the buy template is shown in the lower right corner, beneath the description region. The sell template is used whenever an item is deleted from the table.

USING CHOOSERS

You're probably wondering why choosers are being mentioned at this point in the middle of a discussion about tables. The reason is that choosers are basically half of a table and bear discussion now.

Choosers present the selected item in a way that is very similar to menus. However, when the chooser is triggered to select an item, the chooser behaves almost exactly like a dynamic table. In fact, it's no accident that the "choose form" and "choose table" of a dynamic table have been given those names. When a user selects an item from a chooser, he is essentially using the exact same process that is used when adding an item to a dynamic table.

Under the covers, choosers work very much like dynamic tables, and authors configure choosers in very similar ways. Choosers are assigned a component and Candidate tag expression that are used to filter the available options. For customizing the choose form, they also possess TitleBar and Description scripts. Consequently, the definition of a chooser will look in many ways very similar to the definition of a dynamic table.

CONTROLLING TABLE ROWS AND COLUMNS

MULTI-COLUMN TABLES

Most tables will possess a single column and display as many rows as possible in the space available to them. However, there will be times when you'll want to use tables with multiple columns. For example, if the items being displayed don't need a lot of horizontal space, it may be much better to display them in two or three columns. In the Savage Worlds data files, both Rewards and Resources are perfect candidates for two-column tables.

You can pre-define a table to have multiple columns by setting the "columns" attribute on the table. When a table has multiple columns, the items are sorted so that they increase going downward in the first column, then continue at the top of the second column and increase going downward, then continue at the top of the third column, etc.

Dynamic Control of Columns

You can control the number of columns possessed by a table dynamically from with a Position script. This is accomplished via the "fitcolumns" target reference and is useful when you want to optimize how many columns are displayed based on information you can't know in advance.

The number of columns specified is used as a recommendation when the HL engine triggers the sizing of the table. If a table starts out 400 pixels wide and you specify "fitcolumns" with a value of 2, HL will immediately re-calculate all the sizing details for the table. The width of 400 will be split across two columns, and the template within will have its Position script invoked with an initial width based on the overall width being split.

If you subsequently set the width of the table to 300 pixels, then the previous request for two columns will persist. The table dimensions will be re-calculated, and this time the new width of 300 will be split across two columns. The template will be processed again with the new dimensions.

Dynamic Control of Rows

It's also possible to restrict the number of rows that a table possesses. This can be extremely useful when trying to fit multiple tables into a limit space, and it is accomplished by using the "maxrows" target reference from within a Position script.

Setting the maximum number of rows for a table truncates it to a height that displays no more rows than specified. It is perfectly reasonable for the table to possess fewer rows than specified (e.g. a table with only one item will still only contain and display one item if you set its maximum number of rows to 3).

When fitting multiple tables into a limited space, you can restrict the height of one table to a suitable maximum number of rows. Then you can split the remaining space between the other tables. After positioning the other tables, you can then extend the restricted table to whatever space remains. This way, if the other tables don't use up all the space that you reserved for them, that space can be used by the initially restricted table.

You'll see a number of examples of the above logic used within the starting data files that are provided with the Kit. It's a valuable technique that makes it relatively easy to carve up the available space between multiple tables that can be of wildly varying individual sizes.

Template Sizing Within Tables

Templates are normally sized either by the layout that contains them or by the template itself. When templates are used within tables, the logic remains the same, but there are additional implications involved.

Triggering of Position Script

The Position script of the template is invoked separately for each individual item being displayed. It is also invoked once when the table sizing process is first initiated. During this initial invocation is when the dimensions of the template are actually used by the containing table. No other positioning logic within the script is applied.

During the initial sizing, you can use the "issizing" target reference to detect that state. When initial sizing is being performed, the template need only calculate its height and can bail out without doing anything else. Since any other script logic is going to be thrown away during the sizing operation, there is no need to perform it.

You will see a check of the "issizing" target reference in most table-based templates within the starting data files provided by the Kit. If sizing is occurring, the script bails out as soon as the dimensions are calculated. While not truly necessary, it is recommended that you use the same technique within your own data files.

Height Control

The height of a template is always expected to be controlled by the template when within a table. Most tables contain fixed-height items, where the height of each item is always the same. For these tables, the template dimensions are expected to be established during the special "sizing" invocation of the Position script.

During sizing, there is **no** specific item available to the template. The HL engine passes in a special, dummy item. This dummy item has values of zero for all numeric fields and values of the empty string ("") for all text-based fields. As a result, the template cannot base its height on the contents of an actual item. It must instead determine its size based on general characteristics, such as font heights and other such information.

Once established during the sizing invocation, the height remains fixed. Any subsequent attempts to set the height to a different value are ignored. The exception to this is with variable-height tables, which are only supported for printed output. Within such tables, each template has its height re-calculated for every item that is output.

Width Control

The handling of the width within table-based templates is significantly more interesting than the height. The width passes through many layers and is treated more as a suggestion than anything else when it is passed into the template. When the sizing invocation of the Position is triggered for a template, the width is initialized to a value that is based on sizing requests made upon the table portal itself within the layout. However, the final decision regarding the width is always at the discretion of the template.

Upon entry to the sizing invocation of the Position script, the width is based on two factors. The total width of the table portal drives the overall width available, while the number of columns specified for the table dictates how the overall width is carved up. Using these two values, the template width is initialized to a value that evenly splits up the overall table width into columns.

In most cases, you will simply accept the width value specified by the table. This is achieved by using the default width that is setup for you and not specifying the width yourself. All of the table-based templates within the starting data files provided with the Kit, and even those within the Savage Worlds data files, simply use the default width given by the table.

If you actually set the template width to a new value, the HL engine honors that new width according to a few important rules.

1. If the table has only one column, the overall table width is changed to be based on the new width given by the template.
2. If the table has multiple columns, the new template width is used to determine how many columns can actually fit within the table, **without** increasing the established table width. Once the number of columns is determined, the table width is changed to be exactly the space needed for those columns. Remember that a table will always be sized to contain at least one item, so setting a template width that is wider than the table width will increase the overall width of the table.

The following are a couple of examples of the above rules being applied. Both examples assume that you start with a table that is 400 pixels in width and that has two columns.

1. If the template sets its width to 300, only one column will now fit. Consequently, the table width is set to 300 pixels to correspond with the one column the table contains.
2. If the template sets its width to 125, three columns with now fit. Consequently, the table width is set to 450 pixels to correspond with the three columns the table contains.

In all cases, the template has final authority over the final dimensions used for the table. The HL engine will suggestion a width that is usually appropriate, but the template is free to override that suggestion as it sees fit.

Table Headers and Footers

Headers

To help make positioning easier, tables have a built-in option to include a header. Instead of having to position the header separately from the table itself, the Kit merges them into a single unit. When trying to dynamically fit multiple tables into a fixed amount of space, this integration makes the task substantially easier. As the name would imply, headers always appear above the contents of the table.

Headers can be defined for all tables, and they work very similarly to the add option of dynamic tables. The quick and easy way of adding a header is to define a "HeaderTitle" script. The script specifies the text to use for the header, and you can use encoded text to including custom highlighting and formatting, but you may only have a single line of text. Because it's a script, you readily change the contents of the header based on conditions within the character. The Kit will automatically handle all the mechanics of showing the header if you specify a script.

There is a second way to specify a header. You can define a custom template and associate it with the table for use as the header, along with an object from which the template contents can be pulled. This technique is useful when you want to display more complex information within the header. For example, you may want to put column headings above a table, and that entails defining your own header template. You can see a few examples of this within the World of Darkness data files on the Armory tab.

Footers

Only fixed tables can possess a footer, and they always appear beneath the contents of the table. On fixed tables, the footer takes the place of the add option on dynamic tables. The footer can only be defined via use of a template, and that template works just like the header template described above. The only difference is the placement of the footer relative to the table. The object referenced by the footer template is the same one used by the header - it is shared.

Tables Always Auto-Size

Tables perform very differently from other portals in terms of their sizing. All portals, including tables, are initially sized at the start of all positioning logic. However, tables will continue to automatically size themselves whenever you make changes to them.

Tables always maintain an integral height and width. This means that the table height is always an exact multiple of the height of the items it contains. It also means that the table width is always an exact multiple of the number of columns it contains. No items are ever partially displayed.

Whenever the size of a table is modified, everything is re-calculated for that table. If the specified dimensions are not an exact multiple of the width and/or height of individual items, then the size of the table is automatically **shrunk** to the next lower integral width and/or height. A table will never be larger than the dimensions assigned to it, but will often be smaller. The lone exception to this that a table will always be big enough to contain a single item. If you attempt to size a table to smaller than is required to display a single item, the table will be sized to show one item.

This auto-sizing behavior is critical to keep in mind when using scripts to optimally fit multiple tables within a layout. The downward adjustment of a table's dimensions by one pixel could result in an entire row or column being dropped from the table. As such, any time that you change the size of a table, you must be sure to retrieve its updated dimensions when positioning another portal relative to it.

Keying on Items Within Tables

When working with tables, there will be times where you will want to base sizing and positioning on the number of items within a particular table. This will most often occur when trying to intelligently fit multiple tables into a limited amount of space.

The Kit provides two important mechanisms for determining the items in a table. First, there is the "itemcount" target reference that returns the total number of items within the table. Second, there is the "itemsshown" target reference that returns the number of items that are actually visible within the table.

The utility of the item count is probably obvious. By checking the item count, you can determine that a particular table contains no visible items. When generating printed output, a table with no visible items should generally be completely omitted. To achieve this, you can easily check the number of items in a table and set its visibility to zero if the count is zero.

The number of items shown is equally useful. By comparing the number of items shown to another value, you can make appropriate determinations about what's going on with the table. Most commonly, you will compare the number of items shown against the total number of items in the table. If the values are the same, then you know that the entire contents of the table are visible. When generating printed output, if the two values do not match, then you know that one or more items were not included in the table. This makes it possible to intelligently determine how to lay out the page, since you can quickly determine whether information is omitted and/or will be appearing in a subsequent spillover section of the printout.

PANEL DISPLAY ORDER

All of the panels shown within HL are assigned a specific order by the data file author. This is accomplished via the "order" attribute on each panel. The attribute specifies a numeric value that indicates the sequence in which the panel should be shown. All panels are sorted based on the order attribute they are assigned.

The resulting sorted order is then used as the sequence in which the panels are actually presented. The tabs across the top of the main window are shown in this order.

Summary panels are always shown after the edit panel, in a separate grouping. They are also sequenced in the order dictated by the attribute each is assigned, but they are grouped separately.

USING AUTOMATIC PLACEMENT

In an effort to make things as easy as possible, the Kit provides a mechanism called "automatic placement" that makes positioning certain visual elements significantly easier. Although primarily intended for use within sheets, automatic placement can also be used in various places with on-screen visual elements.

Automatic placement can only be used on visual elements within layouts and scenes. Each of these element types manages internal logic to support automatic placement, so you can use the mechanism whenever it suits your needs. Automatic placement assumes that you are placing a progression of visual elements in a vertical arrangement, with each successive element appearing beneath the previous element. Consequently, the mechanism lends itself best to printed output, but it can also come in handy for on-screen positioning.

HOW IT WORKS

All placement is performed within a rectangular region. Before anything is placed, the bounds of this region are initialized to be the full height and width of the visual container (i.e. the layout or scene). When placement begins, each new placement consumes vertical space within the region. This automatically shrinks the region, moving the top of the region downward to the bottom of each new visual element that is placed.

Automatic placement is performed via the "autoplace" target reference. Each placement can specify a gap that should appear between the new element and the one previously placed, which enables appropriate spacing between visual elements.

Additional target references provide the author with complete control over the bounds of the region within which automatic placement is performed. This makes it possible to place specific elements at the top and/or bottom of the visual container, then adjust the automatic placement region accordingly, and finally perform automatic placement of the remaining visual elements. You can also place elements automatically and then retrieve the bounds of the remaining unused space, after which you can manually place visual elements in that space.

THE RULES

In general, automatic placement is very easy to use and very intuitive in how it handles various situations. However, in the interest of clarity, the following specific rules govern how automatic placement behaves.

1. When an element is automatically placed, the width of that element is set to the width of the automatic placement region for the visual container. In other words, each element is sized to take up the full width of the container.
2. When a layout or template is automatically placed, that element is immediately rendered upon completion. This ensures that the sizing of that element is updated so that the top of the region can be accurately moved to the bottom of the element.
3. When automatically placed, most visual elements have their height set to the full remaining height of the automatic placement region. The following caveats apply:
    a. The lone exception to this rule is when a non-table portal is automatically placed within a layout (e.g. a label). In such cases, the height of the portal is assumed to be whatever default height is initialized.
    b. Since the height is set to the full region during automatic placement, it is assumed that every visual element being placed will properly truncate its height as part of its rendering. For example, a template placed within a layout or a layout placed within a sheet must properly set its height at the end of the Position script, basing the height on the extent of the bottommost item within the element.
    c. Table portals automatically determine their extent, so automatic placement of tables works smoothly, without the need for any special handling.
4. When automatic placement attempts to place a visual element that will not fully fit in the remaining space, the region is considered to be fully utilized and no further elements will be placed.
5. If a table is placed and it does not fully fit in the remaining space, as many items as will fit are output. If the table is within a sheet, all remaining items are treated as "spillover" for output in subsequent tables. If within a panel, the table is assumed to provide scrolling to view the excess items.
6. Any visual element that is not displayed at all within the region is designated as non-visible. This means that any table that contains zero items is declared non-visible. Similarly, any templates and/or layouts that do not fully fit are deemed non-visible.
   IMPORTANT! The key exception to this rule is dynamic and auto tables. Since these tables must allow the user to add items to them at all times, they are always shown even when they contain zero items.
7. Until at least one visual element with actual contents is successfully placed within the region, the gap is always considered to be zero. This ensures that the first item actually placed in the container always starts at the top, regardless of how many elements render no contents

DATA FILE DEVELOPMENT PROCESS

Before you dive in and start writing your own data files, there are a number of important aspects to the overall development process that you should be familiar with. The topics below strive to relay some basic knowledge that will be incredibly helpful as you begin the authoring process.

Enable Data File Debugging

Before you do anything else, make sure that you've configured HL to enable all of the built-in data file development and debugging aids. By default, HL assumes that users are not creating their own data files, so assorted development facilities within the product are disabled. You need to make sure they are turned on so that you can put them to use.

To enable these tools, go to the "Debug" menu within HL and make sure the "Enable Data File Debugging" option is checked. If it's not checked, click on it once to toggle the state.

Data File Compiler

The HL engine includes a compiler that processes all of the data files you create. The benefits of a compiler are two-fold. First of all, the compiler allows HL to convert all of the disparate data files for a game system into a highly optimized version that can be used. This results in significant performance improvements and much lower memory requirements, thereby allowing HL to manage lots of information efficiently on even older, slower computers.

The second big advantage of a compiler is that the compilation process vets the data files that you've written. If there are errors in the data files, they can be caught in advance and reported to you, allowing you to fix them. Without a compiler, you wouldn't know if you had an error until you tried doing something that uncovered the error, thereby making it harder to verify that your data files work flawlessly.

NOTE! Even though HL uses a compiler, there are some kinds of errors that the compiler simply cannot catch. The vast majority of errors will be caught by the compiler, but some will not. These errors will be trapped and reported as run-time errors, and they are discussed separately in the section on debugging data files.

Compiling Data Files

During the course of developing your data files, there will be times where you want to fully test that everything is working the way you want. There will also be times when you simply want to verify that your changes are valid and compile successfully. You can ask HL to re-compile your data files at any time by going to the "Debug" menu and triggering the "Compile Data Files" option. You'll be prompted to specify the game system to re-compile, after which you'll be shown any error messages that might be encountered during the compilation process.

As long as your files fail to compile, they will not load in the HL, so you should get in the habit of frequently re-compiling your data files. This will uncover problems quickly, since the error must exist in whatever changes you've made since the previous successful compile.

As a convenient shortcut, you can use the <Ctrl-C> key combination to invoke a compile. This makes it easy to regularly verify that your data files are valid at each step along the way as you develop them. Please note that the <Ctrl-C> key combination will not work when the input focus is an edit portal, since the <Ctrl-C> is interpreted as a traditional "Copy" command within an edit portal.

Using Quick-Reload

Whenever you make changes to your data files, you'll need to load those changes into HL so that you can use and test them. The obvious way to do this is to go to the "File" menu and select "Switch Game System". However, this approach always shows you the release notes for the game system and potentially the "demo mode" warning, after which you'll be shown the "Configure Hero" form for a new character. After a few dozen times, this process gets really old.

To bypass this, HL includes the "Quick Reload" mechanism, which can be invoked by going to the "Debug" menu and selecting the "Quick Reload" option. This mechanism re-compiles the data files, if necessary, and then reloads them into HL, bypassing the extra steps. It also restores the current tab that is selected. As an added bonus, if you have a saved portfolio loaded, your portfolio is also reloaded. This makes it quick and easy to incrementally modify and test out behaviors associated with selected options.

Take Snapshots Regularly

As you evolve your data files, you will be making significant changes. Even if you are careful, it's likely that you will end up causing everything to break at a few points along the way. When this happens, it can be invaluable to be able to see exactly what has been changed since the last time everything was working fine. In order to do this, you need to have a saved copy of when things were last working. Consequently, we **strongly** encourage you to make a complete copy of your data files at regular intervals, preferably at milestones where everything is working the way you want it. We refer to these copies as "snapshots".

The easiest way to take a snapshot is to use the HLExport tool that is included with the Kit. This tool is designed to package up all of the data files for a game system into a single file that can be easily imported back into HL and you'll be using this tool to distribute your data files once they're created. In the meantime, though, this tool can also be helpful during development. Using HLExport, you can readily take snapshots of your working data files and save them. If you need to refer back to an old snapshot, you can import the file back into HL and compare the files.

IMPORTANT! If you use HLExport as outlined above and need to reload an old snapshot, be sure to import the files into a **different** directory from the data files you are developing. Otherwise, the old files you import will overwrite your recent changes!

Another simple technique is to make a copy of the entire directory contents for your game system. This allows you to do a direct file-to-file comparison of any file at any time, which can be quite handy at times. The only drawback of this approach is that it often requires more effort than the HLExport technique.

In general, a combination of both techniques will often yield the best results.

Skeleton Data Files

To make it as easy as possible to get started writing data files for a new game system, the Kit includes a starting set of data files. These data files aren't just a hodgepodge of examples, though. They are a fully operational foundation that serves as a framework that you can adapt and build upon to create a solution for virtually any game system.

Since this framework provides all of the basics you'll need and must simply be fleshed out, we refer to this starting point as the Skeleton data files. These data files are minimal in nature, but they offer a solid framework to start with, including a variety of built-in mechanisms that you will likely find yourself using for whatever game system you set out to implement.

REVIEW THE SAMPLE DATA FILES

Before you start trying to develop your own data files, you should first spend a little bit of time familiarizing yourself with the Skeleton data files. Since they will form the starting point for your efforts, you'll benefit substantially by understanding how they work. You should also take the time to review the contents of the data files themselves and get familiar with them, as you'll begin modifying and adapting them.

You can see the Skeleton data files in action by looking at the "Authoring Kit Sample" game system that is installed with HL. If you take a look within the "sample" data file folder, you'll see all of the data files for the game system. Lots of useful insights into the sample data files will be found in the section **Exploring the Sample Game System**.

The only differences between the Sample game system and the Skeleton data files is that the Sample game system includes an assortment of attributes, skills, abilities, weapons, and whatnot. By including these objects, you can better see how everything works. In contrast, the Skeleton data files omit these objects, since you would otherwise have to delete them before you could begin adding all the material for your own game system.

Functionality, the two sets of data files are identical. The only difference is that the Sample game system includes a variety of things that you can actually play with.

STUDY THE SAVAGE WORLDS EXAMPLE

In addition to the Skeleton data files, a separate set of data files is provided that implements a reasonably sophisticated game system - the Savage Worlds system from Great White Games. The Kit documentation includes a detailed walk-through of how the Skeleton data files were adapted for that game system, guiding you through the entire evolutionary process. This offers a concrete example of how to approach your own project.

When starting on the development of data files for a new game system on your own, you'll be starting out with the Skeleton data files and expanding upon them for your own purposes. It would be extremely helpful to first review how the Skeleton data files evolve into the Savage Worlds data files, as there will likely be many similarities with the evolutionary process for your game system. The Savage Worlds Walk-Through will also provide insight into how and when to address different steps in the development process that you can leverage within your own project.

HAVE A PLAN

When you set out to write your own data files, there is one detail that is more important than virtually anything else: have a plan. It is critical that you first do your homework and map out both how everything will work internally and how it will all work and behave. If you launch into writing your data files without a solid plan, you will almost certainly run into a substantial number of surprises and setbacks along the way. Heck, you'll likely have a fair number of those even if you do have a good plan in place.

While surprises and setbacks won't stop you from ultimately creating your data files, they will almost certainly cause unnecessary delays and frustration. So you will fare best if you take the time upfront to develop your basic implementation strategy, map out all of the structural pieces you'll need, and design how everything should look and behave for the user.

From a structural perspective, identify all the major elements of the game system. Figure out what types of objects you'll need, and then determine the components and compsets necessary. Anticipate the various game behaviors that you need to properly model and map those into the appropriate fields and tags. Sift through all the facets of the game and you'll likely uncover a lot of subtle details that you'll need to fully implement in your data files. It's amazing how many little details lurk within even the simplest game systems. When it comes time to write data files for the game, you'll need to deal with all of them, so it's much better to create a lengthy laundry list upfront so that you can plan for them and not get hit with lots of surprises along the way.

For the visuals, figure out all the various pieces that the user needs to manage. An excellent place to start is the character sheet, but don't stop with that. There are often lots of little details that publishers don't include on the character and that would be a great benefit to support in your data files. Figure out how you're going to visually organize everything across and within the various tab panels. Draw sketches of each tab panel that show what pieces are involved on each. Assemble a road map for how everything will hang together and how the user will interact with it all.

Once you've got the above tasks complete, you're ready to start writing the data files for your game.

CREATING THE NEW GAME SYSTEM

When you're ready to launch into developing your own data files, we've made it as easy as possible to get started. There are a vast number of details involved in getting a set of data files to a critical mass where you can successfully load them and experiment with them. There are a similarly large number of mechanisms that will be useful for just about any game system. The good news is that we've saved you the work of having to do all this, as the Kit includes the Skeleton data files that will let you hit the ground running.

Creating the framework for a new game system entails only a few mouse clicks. Go to the "Develop" menu and select the "Create New Game System" option. You'll be prompted to enter both the name of the game system and the name of the folder in which to place the data files. Once that's done, HL will set everything up properly for you. After creation, you can immediately switch to your new game system and set it in place, then you're off and running.

WARNING! There are a number of critical implications with the choice of folder name. Please review the list below before selecting the folder:

- Saved portfolios are associated with a game system via the folder assigned. Changing the folder for a game system after the initial release of data files will result in all existing saved portfolios being rendered inaccessible.
- Until the release of your data files to others you may freely change the folder name within the definition file although doing so will invalidate any portfolios you've created thus far.
- Make sure the folder name uniquely identifies the game system. When a user imports the data files, this folder is where the data files will be installed. If two or more game systems use the same folder name, they will overwrite each other.
- The folder name may consist only of alphanumeric characters (i.e. letters and digits). It may not contain any spaces or special characters other than the underscore ('_') and hyphen ('-').

Advanced Authoring Concepts

This section delves into many of the more advanced concepts involved in creating data files. These concepts cover both structural and visual facets of the data files, as well as suggestions for an effective design philosophy. Simply click on one of the many topics below to learn more about it.

IMPORTANT! This section of the documentation is not yet complete. Topics that have been written will be found at the top of the list below and possess live links. Other topics are simply identified by name, sometimes along with a few notes about what the topic will contain. These topics will be added over time to complete the documentation.

- The "Live" State
- Bootstraps
- Automatically Adding Picks to Actors
- Advanced Script Handling

[TBD]

- Actor Rules
- Required Elements
- Just-in-time Information
  - MouseInfo Scripts
  - Description Scripts
- Game System Logo
- Leveraging Usage Pools [!]
- Advanced Things
  - Uniqueness of Things [!]
  - Pick Linkages [!]
  - Replacement of Things
- Validation
  - Detecting validity of structural elements
  - Reporting validation errors to user
  - Highlighting validation errors to user (i.e. color-coding)
  - Errors Versus Warnings
  - Prompting to select missing information
  - Panel Linkages [!]
- Managing Gear [!]
  - Holders
  - Gear Weight Determination [!]
  - Stackable vs. Non-Stackable [!]
  - Splitting and Merging
- Character Advancement [!]
  - Restricting when advancement is allowed
  - Gizmos with dynamic tagexprs provide configurable advances
  - Advances are displaced to actor
  - Unwind mechanism
  - Autonomous Objects
  - Editing of previous advancements
  - Switching back to creation mode
- Pre-Requisites [!]
  - Dependencies on Specific Things [!]
  - Expression-Based Requirements [!]
  - General Pre-Requisites [!]
- Advanced Components and Component Sets [!]
  - Re-using Components in Multiple Component Sets
  - Designing Components for Re-Use
  - Adding "Short Name" Behavior [!]
  - Creation/Deletion Handling
- Advanced Fields [!]
  - Field Types
  - Field Styles
  - Persistence of Fields
  - Bounding of Fields
  - Calculated Fields
  - Finalized Values (Picks vs. Things)
    - Using calculated fields to ensure updates to finalized values
  - Delta for User Fields
  - Formatting of Fields (signed, multiline, integer/float)
  - History Tracking
- Character Sheet Output [!]
  - Standard Sheets
  - Spillover Sheets
- Managing Dossiers
- Statblock Output
- Editor Integration
  - Edit Things
  - Input Things
  - Designing for Editor Integration
- Debugging Techniques
- Distributing Data Files
  - Designing data files for user-extensibility
  - Game System FAQ
  - Appropriate Copyright Information [!]
  - Relying on Minimum Product Features [!]
  - Importance of Release Notes [!]
  - Version Numbers for Data Files [!]
  - Stock Portfolios
  - Using the HLExport Tool [!]
  - Publishing Your Files Through Hero Lab
- Special Tags
  - Actor Tags [!]
  - Identity Tags [!]
- Actor Configuration
  - Sources [!]
    - Required Sources
    - Precluded Sources
  - Rule Sets and Contexts
- Advanced Gizmos [!]
  - Editable Gizmos
  - Associating a Form
  - Setting Up a Default Thing
  - Buy Templates for Gizmos
  - Shadowing [!]
  - Displacement [!]
- Transactions [!]
  - Buy vs. Sell
  - Quantity vs. Cash

    - Using Buy Templates for Customizing Selections (e.g. with choosers)
    - Transaction Tags
- Dashboard
- Tactical Console
    - Initiative
- Design Philosophy
    - Development strategy
    - Designing Data Files
    - Iterative Evolution
- Visual Interactive Behavior
    - Rules for auto-sizing and default sizing of portals
    - Where to use tables, choosers, menus, etc.
    - Showing items within tables
    - Choosing items within tables and choosers
    - Restricting available items within tables and choosers
    - Dual-purpose Headers on Tables [!]
    - Freezing portals
    - Tricks with templates and portals
    - Tricks with layouts, panels, forms, and sheets
    - Automatic placement mechanism for layouts and templates
- Macros
- References
- Condition Tests
- Hidden Things
- Imposing Unwind Logic on Picks [!]
- Minion and Master Relationships [!]
    - Masters Influencing Minions
    - Minions Influencing Masters
- Adaptive Portfolio Loading
    - Load Mods
        - Source Maps
        - Field Maps
        - Silent Objects
    - Load Fixups
- Configuring the Dice Roller [!]
- Evaluation cycle
    - Relative timing of Leads vs. masters vs. minions
    - Rules for sequencing of tasks with the same phase and priority
- Using Visual Resources [!]
    - Built-In Resources
    - Adding Custom Resources
    - Transparent Bitmaps
    - Managing Styles [!]

THE "LIVE" STATE

A variety of objects possess a "live" state. The "live" state controls whether an object **behaves** as if it exists (live) or does not exist (non-live). This makes it possible to automatically include objects within your data files and then dynamically have them appear or disappear, based on conditions within the portfolio that the user is constructing.

The "live" state applies equally to both visual elements and structural elements. A simple example of using the "live" state with visual elements is the various tab panels associated with each different class. All of those tabs are always defined and present, but each appears only when levels of the corresponding class have been added to the character. This is controlled via the "live" state.

For structural elements, the "live" state is primarily used to govern picks. A classic example in the Savage Worlds data files is arcane skills. Arcane skills are only possessed by a character when he has selected the corresponding arcane background. As such, the "live" state is used to control whether the arcane skills are made available.

TAG EXPRESSIONS CONTROL THE STATE

The "live" state of elements is always controlled via a tag expression. The object upon which the tags are tested will vary between different types of elements, but a tag expression is always employed.

The results of the tag expression dictate the "live" state of the object. If the tag expression returns "true", then the object is considered "live". If the tag expression returns "false", the object is considered "non-live".

The tag expression is evaluated at different points in time, which depend on the nature of the element. Whenever the tag expression is re-evaluated, the behavior of the object can transition if the results of the tag expression change. This means that, whenever the tag expression is re-evaluated, the object can immediately transition from "live" to "non-live", or vice versa. The implications of transitioning the live state are different for visual and structural elements, as outlined in the sections that follow.

VISUAL ELEMENTS

The "live" state of visual elements is always controlled via a Live Tag Expression. This tag expression is always applied against the structural element whose contents are being displayed through the visual element. This is typically an actor, although it can also be a gizmo if the showing a form that edits the contents of the gizmo.

The "live" state is verified when the display is updated, which always occurs after each new evaluation cycle. If the container fails the tag expression test, then the visual element is completely hidden. In addition, any Position script for the visual element is ignored, since there should be nothing to show or position within the visual element.

STRUCTURAL ELEMENTS

The "live" state of structural elements is generally governed by a Container Tag Expression, although other mechanisms also exist. These tag expressions are always tested against the tags on the container of the object, hence the name. The object sets forth the requirements that must be met by any prospective container that wants to hold the object.

The "live" state is tested in two separate situations for structural elements. First of all, when a thing is a candidate for display as a selectable item by the user, the "live" state is checked. If the thing fails the test and ends up being non-live, the thing is completely omitted from the list of items shown to the user. If the thing is live, it is shown normally.

Once a thing is added to a container as a pick, the "live" state is again tested as part of every evaluation cycle. When a pick becomes "non-live", it goes dormant. All behaviors associated with the pick are turned off. For example, any scripts that are associated with the pick are simply ignored by the HL engine. The pick is treated as if it were never added in the first place.

However, if a pick becomes "non-live", it cannot simply be automatically omitted from the character. If the user added the pick, then the pick becomes dormant, but some additional behaviors are automatically implemented. First of all, any user-added pick that goes non-live remains visible to the user everywhere. The pick also becomes invalid, which allows for it to be color-highlighted to the user as an error that needs correction. Lastly, a validation error is reported to flag the error to the user.

A structural element that goes non-live has implications upon all elements that are chained to it. This includes all bootstrapped picks, any child gizmo, and any minion that is attached via the pick. Whenever a structural element goes non-live, anything it chains to also goes non-live. Consequently, any chained elements go dormant and simply disappear. The only exception to this is a unique pick that is added via multiple roots (e.g. bootstrapped by two or more picks), in which case the chained pick only goes dormant when all of its roots go non-live.

BOOTSTRAPS

There will be many situations where you want to add a pick to a container automatically. For example, you'll need to make sure that every actor starts out with all of the attributes possessed by characters. You'll also want to re-use abilities, so you'll need to have one thing automatically add another thing.

Within the Kit, this process is referred to as "bootstrapping". Correspondingly, when you add a thing to a container automatically, the resulting pick is referred to as a "bootstrapped" pick. There are a number of important implications associated with bootstrapping, which are discussed in the topics below.

THINGS BOOTSTRAPPING OTHER THINGS

You can bootstrap picks from a variety of situations. However, the most common situation will be when you have one thing bootstrap another. This will regularly occur when the game system has an assortment of abilities that are conferred from a variety of sources. For example, consider the "low-light vision" ability within the d20 System. This ability is conferred by multiple different races, certain magic weapons, etc. You're only going to want to define the ability once, after which you can have each race and weapon simply bootstrap the ability.

When a pick is bootstrapped by another pick, the pick that does the bootstrapping is referred to as the "root" pick. The root pick has complete control of the bootstrapped pick. If the root pick is deleted, the bootstrapped pick is also deleted. If the root pick goes non-live, so does the bootstrapped pick. The bootstrapped pick still has its own independent existence, but that state is wholly dependent upon the root pick as well.

SITUATIONS WHERE YOU CAN BOOTSTRAP

In addition to things bootstrapping each other, there are a number of additional situations in which bootstrapping of things can be performed. These situations are outlined below. In each of these cases, the dependency relationships outlined above for things bootstrapping things do not apply.

- Things can be bootstrapped onto all actors when the actor is initially created.
- Things can be bootstrapped onto entities as part of the entity's definition. The bootstrapped picks are part of every gizmo created from that entity.
- Things can be bootstrapped onto entities when they are attached. The bootstrapped picks are only included on gizmos created by that thing.
- Things can be bootstrapped onto minions when they are attached. The bootstrapped picks are only included on minions created by that thing.

BOOTSTRAPPED PICKS ARE NOT DELETABLE

The topic name pretty much sums it. When a pick is bootstrapped into a container, only the source that performs the bootstrap has control over the pick's existence. Consequently, all bootstrapped picks of fundamentally not deletable. Any attempt to delete a bootstrapped pick will fail.

If you want to pre-select a pick into a table or chooser, and you want the user to be able to delete that selection, you need to use a different technique. Please refer to the section "Automatically Adding Picks to Actors".

BOOTSTRAPPING THE SAME THING MULTIPLE TIMES

At this point, you may be wondering what happens when the same thing is bootstrapped multiple times into a container. The answer depends on whether the thing is designated as being unique. If not, then separate, independent picks are always created, and their behaviors are not linked in any way. However, if the thing is designated as unique, then only one pick is ever added to a given container. This means that all of the bootstrap actions will actually reference the identical pick.

When the first bootstrap occurs, the new pick is created. Each subsequent bootstrap simply increases the reference count for the pick. As long as at least one of the sources for the bootstrap remains in existence, so will the bootstrapped pick. This is important when things bootstrap over things and those things are user-added. Consider the situation where both ThingA and ThingB bootstrap ThingZ. When the user added ThingA, ThingZ is bootstrapped. When the user adds ThingB, the reference count is increased. If ThingA is deleted, the reference count is decreased, but ThingZ still exists. When ThingB is deleted, the reference count goes to zero and ThingZ is finally deleted as well.

When multiple things bootstrap the same, unique thing, their effects are cumulative upon the new pick. If ThingA specifies an auto-tag for ThingZ, and ThingB specifies a different auto-tag for ThingZ, then both auto-tags are assigned to ThingZ. If only ThingA is added to the container, only its auto-tags are assigned, and the same holds for ThingB. However, if both things are added to the container, both auto-tags are assigned.

If multiple things bootstrap the same, unique thing, all of the conditions associated with the root picks must be handled in accordance with some sort of rules. For example, we'll assume ThingA and ThingB both bootstrap ThingZ, and both ThingA and ThingB have been added to the same container. Now we'll further assume that ThingA has a Live test that is currently failed. This means that ThingA is treated as not existing within the container, although ThingB has no dependency and therefore fully exists. So what happens with ThingZ?

To resolve situations like this, the Kit treats each root pick independently. If any one of the root picks for a bootstrapped pick is considered live, then the bootstrapped pick is also considered live. In the example above, this means that ThingZ would be treated as being live due to ThingB being live. The fact that ThingA is not live is irrelevant.

THE MECHANICS OF BOOTSTRAPPING

The process of bootstrapping a thing is typically accomplished via the "bootstrap" element. Since there are a variety of places where bootstrapping can be performed, this element is re-used throughout the Kit.

You can also bootstrap things via the "autoadd" element and the "enmasse" element. Both of those elements are exclusively used within structural files.

COMPONENT BOOTSTRAPS

The vast majority of bootstraps are simple. The bootstrap is always performed for the source context it is defined within. However, bootstraps on components are sometimes an exception. A bootstrap definition on a component will be inherited by all things derived from that component. However, you may only want the bootstrap to be applied to most of those things, with some not having the bootstrap.

In these situations, a Match tag expression can be specified with the bootstrap. This tag expression is applied to each thing derived from the component. Only the things that satisfy the tag expression are assigned the bootstrap. Note that the tag expression is tested against the tags possessed by each thing, so each thing uniquely controls whether it does or does not receive the bootstrap. This also means that only the initial tags each thing possesses are tested.

IMPORTANT! If a component bootstrap does not specify a Match tag expression, the bootstrap is automatically considered a match to all things derived from the component and assigned to all of them.

CONDITIONAL BOOTSTRAPS

There is another potential wrinkle with bootstraps. Some things can be added to different containers or under different circumstances. Depending on the situation, you may want the bootstrap to be added in some cases but omitted in others. For example, a special ability that is selected directly by the user for an actor may behave one way, while that same ability being added as a power within a magic item may behave a bit differently.

To accommodate special cases like this, a Container tag expression may be specified. This tag expression is applied to the prospective container for the new pick. If the container satisfies the condition test, then the bootstrap is added normally. However, if the container does not satisfy the test, no bootstrap is added.

The Condition test on bootstraps works differently from the Match tag expression. In addition to focusing on the container, the test is applied against tags that are dynamically assigned to that container. This means that the Condition test must be scheduled at a specific phase and priority during the evaluation cycle. Any tags that you wish to test for within the container must be handled prior to the timing of the Condition test.

IMPORTANT! All tasks that operate upon a pick must occur after any bootstrap Condition tests for the pick. This includes all component scripts. If a task is determined to be scheduled prior to the Condition test, an error will be reported.

IMPORTANT! Within the Condition test, it is valid to test field values, but the source from which these field values are retrieved will vary. The handling of each possible situation is outlined below:

- If the bootstrap is being added by another pick, all field value tests are applied to the root pick that is performing the bootstrap.
- If the bootstrap is being added by a gizmo, then all field value tests are applied to the anchor pick that attaches the gizmo.
- If the bootstrap is added by a minion, the field value tests are applied to the master pick that attaches the minion.
- For global bootstraps that are applied to every actor, no field value tests may be utilized, as there is nothing to compare against.
- In all cases, the author is responsible for ensuring that only field values that properly exist are tested, since attempts to access missing fields will result in an error being reported.

AUTOMATICALLY ADDING PICKS TO ACTORS

When a new actor is created, it is a virtually empty container. Exactly one thing is automatically added to each actor. It's the "actor" thing that is automatically created for you by the Kit. Other than that, an actor is empty.

So what about all of the various details that are a fundamental part of every actor? Every actor clearly needs to possess all of the basic attributes for the game system (e.g. strength, intelligence, etc.). In most games, the set of skills is pre-set, so those should be part of each actor, too. Every game system has its own unique set of things that should be a basic part of each actor. However, the Kit has no way of knowing what those particular things are for each game system.

To deal with this, the Kit provides three separate mechanisms for specifying which things should be automatically added to every actor. Each mechanism works slightly differently and is intended for use in different situations. Together, they should make it easy for you to pre-configure every actor with all of the picks that it needs. The topics below describe each of these mechanisms.

ADDING INDIVIDUAL THINGS

Each thing that needs to be automatically added to each actor can be explicitly specified. This is accomplished via the "bootstrap" element within a structural file. This element allows you to specify the unique id of the specific thing to be added.

There are quite a few implications associated the bootstrapping a thing into an actor. If you have not yet done so, now would be an excellent time to familiarize yourself with all the particulars of bootstrapping things into containers.

ADDING GROUPS OF RELATED THINGS

As mentioned above, each game has groups of things that should all be added to each actor. Attributes, skills, saving throws, and other traits are perfect examples. Adding these things individually would be both tedious and error prone. It would also make it more

difficult for others to extend the data files by adding their own custom traits, because they would have to manually add any new things they defined.

As long as a group of things can be readily identified by tags they share in common, you have them all automatically added in a single operation. Since all related traits typically derive from the same component, and since every thing possesses a tag for each component it derives from, this requirement is usually a non-issue.

To specify a group of related things that should be added to every actor, use the "enmasse" element within a structural file. This element allows you to specify a tag expression that will identify the related things to be added.

You can also specify tags that will be automatically assigned to each added pick. While rarely needed, this can be useful when the same thing can be also added to the actor additional times by the user.

Pre-Selecting Things Within Tables and Choosers

The above mechanisms allow you to add things to an actor on a fixed basis. This is perfect for attributes and skills, but it doesn't solve all situations. Throughout the interface you design, there will be various tables and choosers where you'll want to pre-select information for the user. In these situations, the user is free to delete or replace the default selection, but one is still provided.

An obvious example is adding an empty character portrait on the Personal tab, since this visually prompts the user to select a portrait of his choice. Another possible situation is when you want to pre-select the contents of a chooser. For example, in the World of Darkness system, the vast majority of characters start out as standard humans, so it's appropriate to pre-select the characters size and speed traits as those of a human. The user is free to change the choices, but he doesn't have to always select them for each new character.

When you come upon a situation where you want to pre-select a thing into a table or chooser, you can use the "autoadd" element within a structural file. This element allows you to specify both the unique id of the thing to be added and the unique id of the portal it is added to.

Any pick that is automatically added to a portal is solely subject to the behaviors for that portal. As such, the pick may not have any Condition tag expression associated with it. In addition, all auto-tags and conditions (e.g. Secondary and Existence) that are normally dictated by the portal are also applied to the pick.

Advanced Script Handling

The Kit provides a variety of more specialized control mechanisms for managing scripts. These mechanisms provide the ability to handle special-case situations that may arise when writing data files for some game systems. The topics below cover these various mechanisms.

Sequencing of Scripts with Identical Timing

When multiple tasks are assigned a common phase and priority, those tasks will be scheduled for evaluation at the exact same time. However, the task scheduler only invokes one task at a time. This means that there would be no guarantee about the order in which two tasks are evaluated that are assigned the same timing. From an authoring standpoint, it's much easier to assign a group of tasks the same phase and priority instead of having to micro-manage which ones occur before each other.

To address this, the Kit provides a set of rules for task scheduling. These rules govern the sequences in which tasks are evaluated that have the same phase and priority. The table below details the order that the Kit uses, with tasks being evaluated in the sequence given, based on their type.

1. Existence tests that are assigned by tables or choosers
2. Bootstrap condition tests that are assigned by root picks
3. Secondary tests that are assigned by tables or choosers
4. Condition tests that are assigned by components
5. Condition tests that are defined explicitly on a thing
6. Calculate scripts defined on any fields
7. Bound scripts defined on any fields
8. Eval scripts that are defined on components or things
9. Evalrule scripts that are defined on components or things
10. Gear scripts that are defined on components

The above rules don't handle a common situation that arises when component scripts are employed. The following topic addresses how this situation is handled.

Sequencing of Component Scripts

When an eval script or evalrule script is defined for a component, all things derived from that component possess the same script. Since the script is assigned a common phase and priority, all instances of that script will be scheduled for evaluation at the exact same time. As discussed above, the task scheduler only invokes one task at a time, so this means that there is no guarantee about the order in which these scripts are evaluated.

In most cases, there is no need to worry about the respective timing of these scripts, since they don't depend on each other. However, there are situations where it's important that all the scripts being evaluated in a guaranteed order. For example, consider the d20 System data files, where attribute bonuses are chosen every four character levels. It's critical that those bonuses be applied in the exact order that they are selected by the user, since those bonuses have ripple effects elsewhere on the character. Consequently, the Kit imposes rules on the evaluation sequencing of component scripts.

Every component must be assigned a "sequence" attribute. This attribute governs how picks are sequenced to the user by default. It also governs how their tasks will be sequenced during evaluation. Consequently, during task scheduling of eval scripts/rules that are associated with the same component, the corresponding tasks are sorted using the sequence assigned to the component. This ensures that tasks are always scheduled in a consistent fashion. It also extends to the case where things are added multiple times to a container.

In rare situations, you'll need to specify an alternate behavior for task scheduling. To accommodate these cases, individual scripts and rules have an attribute that overrides their behavior. This attribute allows to you specify a different component whose sorting rules must be used when scheduling all tasks for that script.

Limiting Evaluation and Reporting

By default, every eval script/rule is scheduled and processed separately for every pick added to an actor. That behavior is exactly

what you'll want 99% of the time, but there are some situations where special handling is needed.

Consider the case where you only want a script to be invoked if a thing is added to the actor. However, what if you also only want the script to be invoked a single time, even if multiples of the thing are added to the actor? There may also be times when you want a rule to be tested for all picks, but you only want the error message to be reported a single time if any of them fail. To deal with these situations, you can specify limits on the evaluation and reporting of scripts and rules.

To limit the evaluation, you can use the "runlimit" attribute to specify the maximum number of times to evaluate the task. This limit is then imposed separately for each container into which the thing is added. If the thing is added ten times to a container, a "runlimit" of one will ensure that the task is only ever invoked a single time. If the thing is added ten times to one container and four times to another container, the task will be invoked once within each container.

To limit the reporting of a rule, you can use the "reportlimit" attribute, which dictates the maximum number of times the rule reports an error. This is critically different from the "runlimit" attribute, which controls the actual invocation. With a report limit, the rule is invoked for every pick, as normal. Only the error message is limited. For example, consider the d20 System data files. If you add multiple class levels to a character, you need to assign hit points for each. You only want to report the error once, but you want all of the picks to be processed so that they will be properly highlighted in red if invalid.

For both evaluation and reporting limits, each thing is normally treated as distinct, with its own limit tracking. Once in a while, you may want to have all things derived from a given component all contribute to the same limit. In other words, the same limit is imposed whether the user adds the same thing multiple times or different things. This behavior is controlled via the "iseach" attribute on components.

Multiple Tasks with Identical Names

When naming scripts and tag expressions for use with timing dependencies, you can assign the same name to multiple tasks. The first question you're probably asking is why you'd even want to do this, so we'll start there. There will be times when you'll have two or more different scripts that all need to calculate the same field, but they do so for different situations.

A simple example is calculating the net attack value for weapons. Melee weapons base the calculation on a "fight" skill, while ranged weapons base the calculation on a "shoot" skill. As a result, you'll have one script for melee weapons and another for ranged weapons. Both calculate the same field, and they should both occur at the exact same time for consistency, so they should possess the same name. That way, scripts that must occur before or after the net attack value calculation don't have to distinguish between whether it's a melee attack or a ranged attack.

When you assign multiple scripts the same name, only one of the scripts is reported in the timing analysis for "errors", "dependencies", and "timing". This is because all identical tasks are assumed to the same for timing purposes, so including them all would be redundant. Identically named tasks are still included in the list of all named tasks.

The drawback of only listing one task is that there is normally no guarantee which task will be chosen by the Kit for use. In most cases, this isn't a problem, but there is one situation where it is. Consider the case where you have two tasks named "MyTask" and various tasks dependent on those tasks. This will work correctly with no difficulties. However, if you then assign a "before" or "after" dependency to one of those two tasks, you run the risk of having that dependency thrown away by the Kit. If the other task is randomly chosen to be kept from the two named tasks, the dependency will be lost.

In this situation, you could always assign the same dependency to both tasks, but that can become a real maintenance headache. What you ideally need is a way to ensure that the Kit properly picks the task that has the additional dependency. This is achieved by designating a particular task as the "primary" task from among a group of tasks with the same name. When a task is "primary", the Kit will always choose that task instead of the others with the same name.

KIT REFERENCE

This section details all of the specific formats and mechanisms used within the Kit. This encompasses all of the different file formats, all the scripting contexts and transitions, required and pre-defined elements, and anything else that requires specification. Click on the various topics below to delve into the details for that facet of the Kit.

REVISION HISTORY

The Authoring Kit is an evolving toolset. A detailed summary of the changes and enhancements made within each new version after the initial V3.0 release is accessible via the links below:

- Functionality Changes and Enhancements
- Skeleton Data File Changes and Enhancements

XML DETAILS

Most of the basics regarding the Kit's use of XML files and their implications can be found in the section on XML Files. Additional reference details are outlined in the topics below.

- XML Character Encoding Set

CONVENTIONS USED BELOW

The reference section of this documentation utilizes a variety of notational conventions for presenting how things work. This includes the syntax used for data files, as well as the formats for other types of files, which are outlined in the topics below.

- XML Attributes in Data Files
- Specifying PCDATA in Data Files
- Optional Attributes in Data Files

TAGS AND TAG EXPRESSIONS

Tags are a fundamental building block that a wide range of mechanisms leverage through the Kit. Tags are utilized to identify and classify objects through tag expressions. The following topics delve into the various facets of using tags.

- Leveraging Tags Via Tag Expressions
- Tag Expression Types

SCRIPTING LANGUAGE

The types of behaviors that exist within the realm of RPGs are limitless. As such, it's impossible for HL to anticipate everything, so the Kit provides a versatile scripting language that enables data files to adapt to virtually any game system. The scripting language has many facets that you should be familiar with, and the topics below outline the information you'll need.

- Scripting Language Overview
- Language Syntax
- Declaring Variables
- Basic Language Mechanisms
- Flow Control
- Other Language Statements
- Language Intrinsics
- Special Symbols
- Script Macros
- Re-usable Procedures
- Debugging Mechanisms
- Compiler Error Messages

SCRIPT DATA ACCESS

The majority of your scripts will focus on accessing and manipulating the data managed within HL. This will involve identifying both the structural and visual elements throughout the data hierarchy. The following topics detail the various pieces involved in data access.

IMPORTANT! Be sure you are familiar with the basics of data manipulation before reviewing this content.

- Script Contexts
- Context Transitions
- Target References
- Data Access Examples
- Employing Script Macros

- Script Types

Structure of Data Files

There are a number of different types of files that comprise the data files for a game system. Each of the topics below describes the structure of one of these file types.

- Definition File Reference
- Structural File Reference
- Data File Reference

Auto-Defined Elements

The Kit automatically defines a variety of different structural and visual elements for use in common situations. These auto-defined elements help to streamline and simplify the authoring process, and they are detailed in the topics below.

- Pseudo-Fields
- Auto-Defined Tags and Tag Groups
- Auto-Defined Components and Fields
    - journal, transact, stackable, gear, shortname, etc.
- Auto-Defined Component Sets
- Auto-Defined Things
- Auto-Defined Sort Sets
- Built-in Resources
- System Resources

Required Elements

There is an assortment of structural and visual elements that every set of data files is required to define. By standardizing on a core set of objects, everything becomes simpler to manage for the data file author. To make the process as easy as possible, the Skeleton data files pre-define these necessary pieces, which you can leave as is or modify if you wish.

- Required Panels
- Required Forms
- Required Components and Fields
- Required Component Sets
- Required Things

Other File Formats

In addition to the various data files, HL utilizes a few other types of files. The contents of these files are documented in the topics below.

- Timing Report File Reference
- Portfolio File Reference

Functionality Revision History

Beginning with the V3.1 release, a summary of the functional changes and enhancements to the Kit within each release is provided below.

V3.1 Revision History

The following changes and additions were introduced in V3.1.

1. Extended "foreach" statement to support "foreach thing in component" syntax, which processes all things derived from a specified component.
2. Extended "foreach" statement to support "foreach bootstrap in thing" which processes all things that are directly bootstrapped by an identified thing.
3. Extended "foreach" statement to support "foreach bootstrap in entity" for use within Description scripts, which processes all things directly attached to the entity associated with the current thing context.
4. The "isentity" target reference on picks and things indicates whether the item has a child entity attached.
5. The "isgizmo" target reference behaves equally for things as well as picks.
6. Picks with attached entities are treated the same as minions with respect to the scheduling of the final condition test. This means the final condition test is scheduled at the time of the latest condition test instead of at the earliest script. This change ensures that a single condition test can be defined on a component so that all child picks can safely test their live state, which depends on the live state of the gizmo, which depends on the live state of the pick that attaches the gizmo.
7. When auto-tags are assigned via a bootstrap with a condition test, the tags are only assigned to the pick when the condition test is satisfied.
8. Bootstraps can selectively override the values assigned to fields within the bootstrapped pick via the "assignval" child element.

9. The "editthing" element can be assigned a "prefix" attribute, which will be used by the Editor to setup an initial unique id for new things created within the Editor.
10. History tracking for fields with "stack" behavior supports the "=" operator, which overwrites the current value with the new value specified.
11. If a field history entry adds or subtracts a negative value, the operator is automatically inverted and the value is negated, changing entries from "+-3" to "-3".
12. The "history" target reference for fields and values supports an optional starting value to be shown in the report. Also, the splicing text can be omitted, in which case ", " is used.
13. The "history" attribute for fields has a "changes" option available that omits any adjustments that yield no actual change, such as "+0" or "*1".
14. When "best" history tracking is used for fields, any adjustments that fail to apply a meaningful change are ignored (like the "changes" behavior above).
15. Portals of type "table_fixed" are automatically omitted by the auto-place mechanism when empty.
16. Field history tracking is now allowed for fields that possess a Finalize script.
17. The "output_dots" portal type was added to make it easy to insert a series of dots between two portals within character sheet output.
18. Header portals used within dual-purpose templates may now use scripts, although the initial script context is undefined and the author must immediately transition to a valid context.
19. The "ischanged" target reference on fields and values returns whether the field value has been changed in some way from its original starting state.
20. The "pushtags" and "pulltags" target references now support containers as either the source or destination, or both.
21. The pseudo-field "thingname" provides access to the original name assigned to a thing, which was otherwise inaccessible if the name of a pick was either modified via a script or renamed by the user.
22. The "scenevalue" target reference provides a mechanism for managing "global" values within the context of a scene to allow communication between the scene and all layout and template scripts within it.
23. The global "state" script context provides support for setting and getting persistent values that are global and exist at the portfolio level, outside of any actor. This mechanism makes it possible to manage state across the entire portfolio.
24. The management of global, persistent, named sets of values is provided within the "global" script context via the "setrandom", "setextract", "setremain", and "setdiscard" target references.
25. The NewCombat and NewTurn scripts possess an "@isfirst" special symbol, which indicates whether the script is being invoked on the very first actor. This allows one-time handling at the start of a combat or turn that spans all actors.
26. The NewCombat and NewTurn scripts are always invoked *before* the Initiative script is invoked for any actor.
27. The "shortname" field is now synthesized at a priority of 100 within the "Render" phase, if that phase exists, else in the last phase of evaluation.
28. The minimum font size supported by "sizetofit" is now 6-point when rendering to the screen and 4-point when rendering onto sheets for printouts.
29. The primary and secondary initiative values for each actor will persist if they are not set to a new value within the Initiative script.
30. The InitFinalize script can be specified in the definition file and will be used as the Finalize script for the "initiative" field.
31. The "initminimum" and "initmaximum" attributes were added to the "behavior" element in the definition file, dictating the lower and upper bounds for the initiative value when the user adjusts the value via the incrementer in the TacCon.
32. The "gaphorz" and "gapvert" target references on visual elements have been renamed to "gapx" and "gapy", respectively, to eliminate confusion regarding their behavior.
33. The various "gap" attributes on table portals have been renamed to eliminate confusion about their use. The "showgaphorz" and "showgapvert" attributes are now "showgapx" and "showgapy", respectively. The "choosegaphorz" and "choosegapvert" were renamed to "choosegapx" and "choosegapy".
34. Printing of a spillover page continues until a page is generated that contains no data to output, at which point printing continues with the next page.
35. Encoded text supports the indenting of the first line of a paragraph via the "{indent value}" syntax. Both normal and hanging indents are supported.
36. Menu styles possess the "droplist" and "droplistoff" attributes, allowing the author to override the bitmaps used for the drop-arrow.
37. The LeadSummary script is allowed to call procedures, which must be either the "container" context or "any" script type.
38. Picks support the "uniqindex" target reference, which returns a value that uniquely identifies the pick throughout the entire portfolio.
39. Only "derived" fields may utilize a Calculate script.
40. The "output_separator" portal can be used within sheet output to insert a solid black line within the dimensions given.
41. The "isroot" target reference (of picks) is accessible from the "template" script context.
42. If the "@message" special symbol is set within an Eval Rule, but the "@summary" symbol is not, the returned message text is automatically used as the summary for display on the validation summary bar.
43. The "exprreq" and "pickreq" elements on things support the "onlyonce" and "issilent" attributes, exactly the same way as they work on standard "prereq" elements.
44. The "image_literal" portal supports the "isbuiltin" attribute, which identifies a bitmap that is provided by HL for general use.

V3.2 REVISION HISTORY

The following changes and additions were introduced in V3.2.

1. The dossier being output adds a "dossier.<id>" global tag to the portfolio during output. This allows scripts to check which dossier is being output.
2. An "EndCombat" script is invoked on each actor when combat ends.
3. An "endcombat" procedure type is supported for use with the EndCombat script.
4. The "meta" text encoding allows the alignment behavior of bitmaps to be controlled relative to the position of text when mixing text and bitmaps.
5. The scripting language supports the "doneif" statement to simplify coding.
6. The "state" script context supports the "thing[id]" transition to directly access all facets of a thing.
7. The "modify" target reference for fields supports the "#" operator, which suppresses the display of any operator within the resulting history report.
8. The "modify" target reference for fields supports the "$" operator, which inserts a history entry with no field value and only a text description.
9. Things possess the "holdlimit" tag expression to restrict the types of things that can be held by the thing within the gear containment hierarchy. This makes it possible to assign laser sights and such to weapons.
10. The "it_field" input thing has an optional Boolean "multiline" attribute to support the entry of multi-line text fields within the Editor.
11. Thing-based menus can specify an alternate field to display for the menu item via the "namefield" attribute.
12. The "heromatch" target reference on picks and containers behaves the same as "tagmatch", except that the initial context is always assumed to be the current actor. This allows arbitrary matches against the hero from anywhere.
13. Sources possess both the "maxchoices" and "minchoices" attributes. These allow an author to specify a minimum and/or maximum number of child sources that can be selected by the user.
14. Added the "plaintext" intrinsic function to the scripting language.
15. Added the "amendthing" target reference to the "thing" script context, which allows pertinent feats in 4E to directly modify the powers that they impact.

SKELETON DATA FILE REVISION HISTORY

The Skeleton Data Files were initially released in V3.0 and have evolved since that time. If you started work on your data files with an older release than is currently available, you should probably integrate the changes listed below for each subsequent release.

V3.1 REVISION HISTORY

IMPORTANT! A detailed list of the specific changes made to each file can be found here.

The following changes and additions were introduced in V3.1.

1. Added highlighting of any currently equipped weapon and/or armor on the Armory summary panel.
2. Eliminated use of the "mousepos" attribute within MouseInfo scripts whenever it specified the default behavior was to be used.
3. Fixed a bug in the standard character sheet that failed to handle weapons that are non-stackable.
4. Integrated use of the "output_dots" portal within the sample character sheet framework that is provided.
5. Added a few lines of missing code to Position script for the "oAdjPick" template.
6. Increased the "maxfinal" length of "grStkName" field within "Gear" component to 100.
7. Fixed problem with the color highlighting not showing red properly in Finalize script for "resAddItem" field of "Resource" component.
8. Revised the gear template to be more intelligent and adaptive.
9. Eliminated some extraneous code from the "power" portal within the "dashboard" template.
10. Fixed a problem in the "DshActive" procedure that was not properly outputting activated abilities that weren't in-play adjustments.
11. Added the "lblSmlLeft" style for general use.
12. Utilized the "lblSmlLeft" style within the TacCon.
13. Fixed problems with the Eval script for generating the name within the "Adjustment" component.
14. Corrected the behavior of the Label script for the "summary" portal within the "tacPick" template of the Tactical Console.
15. Cleaned up the code in various scripts within the "tacPick" template of the Tactical Console.
16. Renamed the "damage" and "status" portals within the "tacPick" template of the Tactical Console to "status1" and "status2", respectively.
17. Renamed the "column1" portal within the "tacPick" template of the Tactical Console to "traits".
18. Renamed the "DashTacCon.Column1" tag to "Traits".
19. The contents of the "peace" and "summary" portals within the "tacPick" template of the Tactical Console are sorted by name for display.
20. The "summary" portal uses "sizetofit" to shrink a bit whenever its contents are too large to fit in the available space.
21. The "static" form includes built-in handling for accessing the master when a minion is being edited.
22. Defined the "actMaster" style and added the corresponding bitmaps as built-in resources.

23. The "Domain" component automatically integrates the domain into the name of the pick, plus the "shortname" field if one exists.
24. The form for advancements uses the base "thing" name when synthesizing a name for display instead of the normal name, providing full control over how the domain is shown for advancements.
25. Suitable abbreviations are now defined for all traits.
26. Revised the logic for shrinking text on the advancements form.
27. Revised the logic for limiting line height on advancements form.
28. Eliminated use of "textheight" for validation report sizing in the character sheet.
29. Revised a number of Position scripts within the character sheet to make proper use of "sizetofit" (some original uses were ineffectual).
30. Fixed problem where the Journal tab was showing the total XP based on the contents of the usage pool instead of the "resXP" resource.
31. Heroes track their "dead" state, which is shown on the TacCon.
32. Added pre-defined "menuSmall" style for smaller menus.
33. Added built-in support for handling things that require special user-selection behaviors, including checkboxes, array-based menus, and thing-based menus. This includes the "UserSelect" component for behaviors and "UserSelect" template to handle visuals.
34. Added built-in thing for handling natural armor, which requires the "defDefense" field to be "derived" instead of "static".
35. Added handling to the "SimpleItem" template that automatically changes the color of any non-deletable items to indicate that state.
36. If an equippable item possesses the "Equipment.Natural" tag at the time of creation, the item is initialized to the equipped state but may thereafter be changed.
37. Added option for centering the name within the "SimpleItem" and "LargeItem" templates.
38. Added the "Equipment.StartEquip" tag and handling for it within the Creation script of the "Equippable" component.
39. Revised the validation rules within the "Domain" component so that any panel linked to the thing is properly highlighted when the domain is required and not specified.
40. The "Domain" component supports customization of the term shown to the user for the domain within validation errors via assignment of a tag from the "DomainTerm" tag group.
41. The advancements mechanism supports the "DomainTerm" behaviors when displaying options that leverage domains.
42. Moved the user manual into the "docs" folder beneath where the data files are stored to keep it separate from the other data files.
43. Abilities marked as "creation only" now verify the appropriate state via the "state.iscreate" test.
44. Revised timing of setting the delta for the "trtUser" field in the file "traits.str".
45. Revised timing of the scripts that tally attribute and skill points within "traits.str".
46. Added a "notation" advance to the advancements mechanism.
47. Sheet output of abilities must be limited to a single line high.
48. Changed the timing of the script handling auto-equipped gear, as it was scheduled much earlier than necessary and limiting authoring options.
49. Moved a variety of bitmaps into the "builtin" folder, which then required changing most "image_literal" portals to utilize the bitmaps there.
50. Added an assortment of named color resources that are then used within the various styles, which allows for easier re-use and replacement by an author.
51. The Finalize script of the "acPPSumm" field on the "Actor" component must show the "trkUser" field instead of "trtLeft".
52. Revamped the starting HTML file for use as a User Manual.
53. The ranged weapons table "arRange" must reference the "Gear" component so that the appropriate transaction scripts are invoked.

V3.2 REVISION HISTORY

IMPORTANT! A detailed list of the specific changes made to each file can be found here.

The following changes and additions were introduced in V3.2.

1. Fixed bug where character sheet output that continued onto subsequent pages did not properly position the right-side column of output.
2. Added use of the "prefix" attribute on "editthing" elements within the Editor.
3. Unspent resources report validation warnings and highlight the appropriate tabs unless assigned the "Helper.NoMinimum" tag.
4. Fixed bug where the "lot cost" of a piece of gear was being calculated incorrectly.
5. Added new system resources that authors can override when tailoring the interface for a custom game system.
6. Fixed bug where ammunition was not having its quantity setup properly for tracking on the In-Play tab.
7. Eliminated the edit portal length from three edit portals that exceeded the maximum allowed length of the underlying field.
8. Fixed problem with the handling of abilities, where they were not being properly flagged on the actor for indication to the user.
9. Added support for entering "Ammunition" via the Editor.
10. Added support for the "Lot Cost" and "Weight" fields for all types of equipment within the Editor.
11. Added a shell "editor.htm" file that can be used as the basis for providing suitable documentation for adding content to the game system via the Editor.

SKELETON FILE CHANGES V3.1

FILE: "DEFINITION.DEF"

Line 24: Eliminate the "manualroot" attribute so that the manual location uses the default.

Line 46: Replace the "required" attribute with "3.1" instead of "3.0".

FILE: "TAGS.1ST"

Line 52: Insert the new tag shown below.

```
<value id="NoAutoName"/>
```

Line 65: Insert the new tag shown below.

```
<value id="Dead"/>
```

Line 79: Rename the "Column1" tag to "Traits".

Line 90: Insert the new tag shown below.

```
<value id="StartEquip" name="Gear starts out equipped"/>
```

Line 245: Insert the new tag group definition shown below.

```
<group
 id="DomainTerm"
 dynamic="yes">
 <value id="Domain"/>
 </group>
```

Line 254: Insert the new tag shown below.

```
<value id="CenterName"/>
```

Line 256: Insert the new tag group definitions shown below.

```
<group
 id="ChooseSrc1"
 visible="no">
 <value id="Thing" name="All Things"/>
 <value id="Container" name="All Picks on Container"/>
 <value id="Hero" name="All Picks on Hero"/>
 </group>
<group
 id="ChooseSrc2"
 visible="no">
 <value id="Thing" name="All Things"/>
 <value id="Container" name="All Picks on Container"/>
 <value id="Hero" name="All Picks on Hero"/>
 </group>
```

FILE: "ADVANCEMENT.CORE"

Line 19: Added the "Notation" tag shown below.

```
<value id="Notation"/>
```

Line 160: Insert new logic to handle notation advancements at the start of the Eval script, as shown below.

```
~if this advancement has an annotation, there is no user-selection, so build
~the name from our pieces and we're done
if (tagis[Advance.Notation] <> 0) then
  perform gizmo.child[advDetails].setfocus
```

```
  field[livename].text = field[name].text & ": " & focus.field[advUser].text
  done
  endif
```

FILE: "COMPONENTS.CORE"

Line 273: Change the reference to the "component.Ability" tag to "component.shortname".

Line 278: Change the reference to the "component.Ability" tag to "component.shortname".

Line 283: Insert the code below to use the standard name if no short name is defined.

```
~if we don't have a short name, just use the regular name
if (empty(short) <> 0) then
  short = name
  endif
```

Line 337: Insert the new "UserSelect" component definition that is provided below.

```
<component
  id="UserSelect"
  name="User Selection">
  <!-- Text to display with the checkbox
          NOTE! If this field is empty, it means NO checkbox is shown for the pick.
  -->
  <field
    id="usrChkText"
    name="Checkbox Text"
    type="derived"
    maxlength="100">
    </field>
  <!-- Indicates whether the checkbox is selected is not -->
  <field
    id="usrIsCheck"
    name="Checked?"
    type="user"
    minvalue="0"
    maxvalue="1">
    </field>
  <!-- Label associated with the first thing-based menu -->
  <field
    id="usrLabel1"
    name="Thing Menu Label #1"
    type="static"
    maxlength="50">
    </field>
  <!-- Tracks the first selection made when a menu choice is required -->
  <field
    id="usrChosen1"
    name="Chosen Thing / Pick #1"
    type="user"
    style="menu">
    </field>
  <!-- Candidate tagexpr used to determine which picks/things are shown in menu #1
            NOTE! If this field is empty, it means NO first menu is shown for the pick.
  -->
  <field
    id="usrCandid1"
    name="Candidate TagExpr for Menu #1"
    type="derived"
    maxlength="1000"
    defvalue="">
    </field>
  <!-- Source to pull choices from within menu #1 -->
  <field
    id="usrSource1"
    name="Source for Menu #1 Choices"
    type="derived">
    <!-- Determine the source to choose from based on the tag, defaulting to "Hero" -->
    <calculate phase="Render" priority="10000"><![CDATA[
      if (tagis[ChooseSrc1.Thing] <> 0) then
        @value = 0
```

```
    elseif (tagis[ChooseSrc1.Container] <> 0) then
      @value = 1
    elseif (tagis[ChooseSrc1.Hero] <> 0) then
      @value = 2
    else
      @value = 2
      endif
    ]]></calculate>
  </field>
<!-- Label associated with the second thing-based menu -->
<field
  id="usrLabel2"
  name="Thing Menu Label #2"
  type="static"
  maxlength="50">
  </field>
<!-- Tracks the second selection made when a menu choice is required -->
<field
  id="usrChosen2"
  name="Chosen Thing / Pick #2"
  type="user"
  style="menu">
  </field>
<!-- Candidate tagexpr used to determine which picks/things are shown in menu #2
            NOTE! If this field is empty, it means NO second menu is shown for the pick.
-->
<field
  id="usrCandid2"
  name="Candidate TagExpr for Menu #2"
  type="derived"
  maxlength="1000"
  defvalue="">
  </field>
<!-- Source to pull choices from within menu #2 -->
<field
  id="usrSource2"
  name="Source for Menu #2 Choices"
  type="derived">
  <!-- Determine the source to choose from based on the tag, defaulting to "Hero" -->
  <calculate phase="Render" priority="10000"><![CDATA[
    if (tagis[ChooseSrc2.Thing] <> 0) then
      @value = 0
    elseif (tagis[ChooseSrc2.Container] <> 0) then
      @value = 1
    elseif (tagis[ChooseSrc2.Hero] <> 0) then
      @value = 2
    else
      @value = 2
      endif
    ]]></calculate>
  </field>
<!-- Label associated with the array-based menu -->
<field
  id="usrLabelAr"
  name="Array Menu Label"
  type="static"
  maxlength="50">
  </field>
<!-- Array of text items user can select from
            NOTE! If the 0th element is empty, it means NO menu is shown for the pick.
-->
<field
  id="usrArray"
  name="Array of Items to Choose"
  type="derived"
  style="array"
  arrayrows="10"
  maxlength="30">
  </field>
<!-- Item selected from the array -->
<field
  id="usrSelect"
  name="Selected Item in Array"
  type="user"
  maxlength="30">
```

```
  </field>
<!-- Initialize the current array-based selection from the array if not defined -->
<creation><![CDATA[
  if (empty(field[usrArray].arraytext[0]) = 0) then
    if (field[usrSelect].isempty <> 0) then
      field[usrSelect].text = field[usrArray].arraytext[0]
      endif
    endif
  ]]></creation>
<!-- Integrate the various user selections into the name of the pick
                NOTE! Must be scheduled after the "shortname" field is synthesized at Render/100.
-->
<eval index="1" phase="Render" priority="500"><![CDATA[
~if we're not supposed to auto-amend the name for this pick, we're done
if (tagis[User.NoAutoName] <> 0) then
  done
  endif
~if we have thing-based menus, determine the text to append to the name
var choices as string
if (field[usrCandid1].isempty = 0) then
  if (field[usrChosen1].ischosen <> 0) then
    choices = field[usrChosen1].chosen.field[name].text
  else
    choices = "-Choose-"
    endif
  if (field[usrChosen2].ischosen <> 0) then
    choices &= ", " & field[usrChosen2].chosen.field[name].text
    endif
~if we have an array-based menu, determine the text to append
elseif (empty(field[usrArray].arraytext[0]) = 0) then
  choices = field[usrSelect].text
~if we have a selected checkbox, determine the text to append
  elseif (field[usrChkText].isempty = 0) then
    if (field[usrIsCheck].value <> 0) then
      choices = field[usrChkText].text
      endif
    endif
  ~if we have no text to append, we're done
  if (empty(choices) <> 0) then
    done
    endif
  ~add the selection to both the livename and shortname (if present) fields
  field[livename].text = field[name].text & ": " & choices
  if (tagis[component.shortname] <> 0) then
    field[shortname].text &= " (" & choices & ")"
    endif
  ]]></eval>
<!-- Report a validation error if no selection has been made for a menu selection -->
<evalrule phase="Validate" priority="10000" message="You must choose an option" summary="Choose!"><![CDATA[
  ~determine the number of menus that NEED selection
  ~Note: Remember that a non-empty tagexpr field indicates menu selection is used.
  var needed as number
  needed = !field[usrCandid1].isempty + !field[usrCandid2].isempty
  needed += !empty(field[usrArray].arraytext[0])
  ~determine the number of menus that HAVE selections
  var actual as number
  if (field[usrCandid1].isempty = 0) then
    actual += field[usrChosen1].ischosen
    endif
  if (field[usrCandid2].isempty = 0) then
    actual += field[usrChosen2].ischosen
    endif
  if (field[usrSelect].isempty = 0) then
    actual += 1
    endif
  ~if the user has chosen something whenever required, we're valid
  if (actual >= needed) then
    @valid = 1
    done
    endif
  ~mark any associated tab as invalid
  if (ispanel <> 0) then
    linkvalid = 0
    endif
]]></evalrule>
```

```
</component>
```

Line 400: Add the new Eval script below to the "Domain" component.

```
<eval index="1" phase="Render" priority="500"><![CDATA[
  ~if we don't need a domain, there's nothing to do
  if (tagis[User.NeedDomain] = 0) then
    done
    endif

  ~if we're not supposed to auto-amend the name for this pick, we're done
  if (tagis[User.NoAutoName] <> 0) then
    done
    endif

  ~if we don't have a domain, use a placeholder for it
  var domain as string
  if (field[domDomain].isempty = 0) then
    domain = field[domDomain].text
  else
    domain = "????"
    endif

  ~add the domain to both the livename and shortname (if present) fields
  field[livename].text = field[name].text & ": " & domain
  if (tagis[component.shortname] <> 0) then
    field[shortname].text &= " (" & domain & ")"
    endif
  ]]></eval>
```

Line 401: Replace the original Eval script with the new one shown below.

```
<evalrule index="1" phase="Validate" priority="9000" message="????"><![CDATA[
  ~if no domain is needed or the domain is specified, we're valid
  if (tagis[User.NeedDomain] = 0) then
    @valid = 1
  elseif (field[domDomain].isempty = 0) then
    @valid = 1
    endif

  ~if we're valid, get out of here
  if (@valid <> 0) then
    done
    endif

  ~if we have a linked panel, flag it as invalid
  if (ispanel <> 0) then
    linkvalid = 0
    endif

  ~synthesize an appropriate message using the correct domain term
  var term as string
  term = tagnames[DomainTerm.?]
  if (empty(term) <> 0) then
    term = "Domain"
    endif
  @message = term & " must be specified"
  ]]></evalrule>
```

FILE: "ACTOR.STR"

Line 163: Change the reference to the "trtLeft" field to "trtUser".

Line 221: Replace the Eval script with the revised script below.

```
<eval index="1" phase="Final" priority="1000"><![CDATA[
  ~if no damage has been incurred, assign a tag to indicate that state
  if (field[acHPNow].value >= field[acHPMax].value) then
    perform hero.assign[Hero.NoDamage]
```

```
  ~if the hero is dead or otherwise out of combat, indicate that state
  elseif (field[acHPNow].value = 0) then
    perform hero.assign[Hero.Dead]
    endif
  ]]></eval>
```

FILE: "EQUIPMENT.STR"

Line 27: Increase the "maxfinal" value from "50" to "100".

Line 219: Insert the Creation script shown below.

```
<creation><![CDATA[
  ~if this is natural equipment, initialize the equipped state
  if (tagis[Equipment.Natural] <> 0) then
    field[grIsEquip].value = 1
    endif

  ~if this equipment is supposed to start out as equipped, initialize the state
  if (tagis[Equipment.StartEquip] <> 0) then
    field[grIsEquip].value = 1
    endif
  ]]></creation>
```

Line 229: Change the Eval script priority from "1000" to "4000".

Line 584: Change the field type from "static" to "derived".

FILE: "MISCELLANEOUS.STR"

Line 103: Insert the new lines of code below into the Finalize script.

```
    @text = "{text ff0000}"
elseif (unspent > 0) then
```

FILE: "TRAITS.STR"

Line 50: Change the timing of the Bound script, assign it a name, and establish a "before" dependency, as shown below.

```
<bound phase="Traits" priority="1000" name="Bound trtUser">
  <before name="Calc trtFinal"/><![CDATA[
  @minimum = field[trtMinimum].value
  @maximum = field[trtMaximum].value
  ]]></bound>
```

Line 97: Change the timing of the Eval script from "9999999" to "5000".

Line 130: Change the timing of the Eval script and setup "before" and "after" dependencies, as shown below.

```
<eval index="2" phase="Traits" priority="10000">
  <before name="Calc resLeft"/>
  <after name="Bound trtUser"/><![CDATA[
  hero.child[resCP].field[resSpent].value += (field[trtUser].value - 1) * 7
  ]]></eval>
```

Line 175: Change the timing of the Eval script and setup "before" and "after" dependencies, as shown below.

```
<eval index="2" phase="Traits" priority="10000">
  <before name="Calc resLeft"/>
  <after name="Bound trtUser"/><![CDATA[
```

Line 253: Change the test from keying on the usage pool to the actual mode, as shown below.

```
  ~if the mode is creation, we're valid
```

```
if (state.iscreate <> 0) then
  @valid = 1
  Done
  endif
```

FILE: "STYLES_OUTPUT.AUG"

Line 217: Insert the new style shown below.

```
<style
  id="outDots">
  <style_output
    textcolor="202020"
    font="ofntnormal"
    alignment="left">
    </style_output>
  </style>
```

FILE: "STYLES_UI.AUG"

Line 236: Insert the new resource definition shown below.

```
<resource
  id="fntmenusm">
  <font
    face="Arial"
    size="30"
    style="bold">
    </font>
  </resource>
```

Line 268: Insert the new color resource definitions shown below.

```
<!-- color used for normal text throughout the ui -->
 <resource
  id="clrnormal">
  <color
    color="f0f0f0">
    </color>
  </resource>

<!-- color used for text on the static panel - not quite as bright -->
 <resource
  id="clrstatic">
  <color
    color="d2d2d2">
   </color>
  </resource>

<!-- color used for text in title labels -->
 <resource
  id="clrtitle">
  <color
    color="c0c0c0">
   </color>
  </resource>

<!-- color used for names of automatically added picks -->

 <resource
  id="clrauto">
  <color
    color="99efed">
   </color>
  </resource>

<!-- color used for disabled text -->
 <resource   id="clrdisable">
```

```
    <color
      color="808080">
      </color>
    </resource>

<!-- color used for summary text that should be a little dimmer than normal -->
 <resource
   id="clrsummary">
   <color
     color="a0a0a0">
     </color>
   </resource>

<!-- color used for bright text -->
 <resource
   id="clrbright">
   <color
     color="ffff88">
     </color>
   </resource>

<!-- color used for warning text -->
 <resource
   id="clrwarning">
   <color
     color="ff0000">
     </color>
   </resource>
<!-- color used for prompt text inviting the user to change something -->
 <resource
   id="clrprompt">
   <color
     color="ffff00">
     </color>
   </resource>

<!-- color used for the 'buy for free' checkbox on buy / sell panels -->
 <resource
   id="clrchkfree">
   <color
     color="a8a800">
     </color>
   </resource>

<!-- color used for text on summary panels - a little dimmer than normal -->
 <resource   id="clrsummtxt">
   <color
     color="d0d0d0">
     </color>
   </resource>

<!-- color used for labels when choosing advancements -->
 <resource
   id="clradvance">
   <color
     color="ffffff">
     </color>
    </resource>

 <!-- color used for text on action buttons -->
  <resource
    id="clraction">
    <color
      color="000088">
      </color>
    </resource>

 <!-- colors used in edit controls -->
  <resource
    id="clredittxt">
    <color
      color="d2d2d2">
      </color>
    </resource>
  <resource
```

```
    id="clreditbck">
    <color
      color="000000">
      </color>
    </resource>

  <!-- colors used in menu and chooser controls -->
   <resource
    id="clrmenutxt">
    <color
      color="84c8f7">
      </color>
    </resource>
   <resource
    id="clrmenuslt">
    <color
      color="1414f7">
      </color>
    </resource>
   <resource
    id="clrmenubck">
    <color
      color="2a2c47">
      </color>
    </resource>
```

Numerous Places: Change the style to reference a named color instead of a literal color, switching from a "textcolor" attribute to a "textcolorid" attribute. The list of changes is given below.

- Line 282 - "clrtitle"
- Line 293 - "clrstatic"
- Line 303 - "clrnormal"
- Line 313 - "clrnormal"
- Line 323 - "clrnormal"
- Line 335 - "clrauto"
- Line 347 - "clrdisable"
- Line 359 - "clrbright"
- Line 371 - "clrwarning"
- Line 384 - "clrprompt"
- Line 394 - "clrnormal"
- Line 404 - "clrdisable"
- Line 414 - "clrprompt"
- Line 424 - "clrwarning"
- Line 434 - "clrnormal"
- Line 444 - "clrdisable"
- Line 454 - "clrwarning"
- Line 464 - "clrnormal"
- Line 474 - "clrdisable"
- Line 484 - "clrsummary"
- Line 494 - "clrnormal"
- Line 504 - "clrnormal"
- Line 514 - "clrsummtxt"
- Line 524 - "clrdisable"
- Line 534 - "clrsummary"
- Line 545 - "clrsummary"
- Line 556 - "clradvance"
- Line 568 - "clradvance"
- Line 582 - "clrnormal"
- Line 596 - "clredittxt"
- Line 597 - "clreditbck"
- Line 607 - "clredittxt"
- Line 608 - "clreditbck"

- Line 618 - "clredittxt"
- Line 619 - "clreditbck"
- Line 624 - "clrdisable"
- Line 635 - "clrnormal"
- Line 652 - "clrnormal"
- Line 669 - "clrnormal"
- Line 732 - "clrnormal"
- Line 805 - "clraction"
- Line 836 - "clraction"
- Line 867 - "clrnormal"
- Line 901 - "clraction"
- Line 937 - "clraction"
- Line 971 - "clraction"
- Line 981 - "clraction"
- Line 991 - "clraction"
- Line 1025 - "clraction"
- Line 1059 - "clraction"
- Line 1085 - "clrnormal"
- Line 1119 - "clrnormal"
- Line 1145 - "clrnormal"
- Line 1171 - "clrnormal"
- Line 1205 - "clrnormal"
- Line 1231 - "clrnormal"
- Line 1265 - "clrnormal"
- Line 1299 - "clrnormal"
- Line 1333 - "clrnormal"
- Line 1359 - "clrnormal"
- Line 1385 - "clrnormal"
- Line 1411 - "clrnormal"
- Line 1445 - "clrnormal"
- Line 1471 - "clraction"
- Line 1497 - "clraction"
- Line 1531 - "clraction"
- Line 1557 - "clraction"
- Kine 1583 - "clraction"
- Line 1609 - "clraction"
- Line 1643 - "clraction"
- Line 1677 - "clraction"
- Line 1711 - "clraction"
- Line 1745 - "clraction"
- Line 1779 - "clraction"
- Line 1813 - "clraction"
- Line 1847 - "clraction"
- Line 1881 - "clraction"
- Line 1915 - "clraction"
- Line 1949 - "clraction"
- Line 1983 - "clraction"
- Line 2017 - "clrnormal"
- Line 2051 - "clrnormal"
- Line 2085 - "clrnormal"
- Line 2119 - "clrnormal"
- Line 2153 - "clrnormal"
- Line 2206 - "clrdisable"
- Line 2232 - "clrnormal"
- Line 2241 - "clrwarning"

- Line 2252 - "clrdisable"
- Line 2261 - "clrchkfree"
- Line 2270 - "clrnormal"
- Line 2311 - "clrnormal"
- Line 2358 - "clrmenutxt"
- Line 2359 - "clrmenubck"
- Line 2360 - "clrmenuslt"
- Line 2361 - "clrnormal"
- Line 2371 - "clrmenutxt"
- Line 2372 - "clrmenubck"
- Line 2373 - "clrmenuslt"
- Line 2374 - "clrnormal"
- Line 2375 - "clrwarning"
- Line 2386 - "clrmenutxt"
- Line 2387 - "clrmenubck"
- Line 2397 - "clrwarning"
- Line 2398 - "clrmenubck"

Line 470: Insert the new style definition shown below.

```
<!-- slightly smaller label that is left-aligned -->
 <style    id="lblSmlLeft">
  <style_label
   textcolorid="clrnormal"
   font="fntsmall"
   alignment="left">
   </style_label>
  </style>
```

Line 1467: Insert the new style definition shown below.

```
<!-- Style used on the master button -->
 <style
  id="actMaster">
  <style_action
   textcolorid="clrnormal"
   font="fntactsml"
   up="actmastup" down="actmastdn" off="actmastup">
   </style_action>
  <resource
   id="actmastup"
   isbuiltin="yes">
   <bitmap
    bitmap="master_up.bmp"
    istransparent="yes">
    </bitmap>
   </resource>
  <resource
   id="actmastdn"
   isbuiltin="yes">
   <bitmap
    bitmap="master_down.bmp"
    istransparent="yes">
    </bitmap>
   </resource>
  </style>
```

Line 2380: Insert the the new style definitions shown below.

```
<!-- small menu portal -->
 <style
  id="menuSmall"
  border="sunken">
  <style_menu
   textcolorid="clrmenutxt"
   backcolorid="clrmenubck"
```

```
    selecttextid="clrmenuslt"
    selectbackid="clrnormal"
    font="fntmenusm"
    droplist="menuarrsm"
    droplistoff="menuoffsm">
    </style_menu>
   <resource
    id="menuarrsm"
    isbuiltin="yes">
    <bitmap
     bitmap="menu_small_arrow.bmp">
     </bitmap>
    </resource>
   <resource
    id="menuoffsm"
    isbuiltin="yes">
    <bitmap
     bitmap="menu_small_arrow_off.bmp">
     </bitmap>
    </resource>
   </style>

 <!-- small menu portal with coloring to indicate contents are in error -->
  <style
   id="menuErrSm"
   border="sunken">
   <style_menu
    textcolorid="clrmenutxt"
    backcolorid="clrmenubck"
    selecttextid="clrmenuslt"
    selectbackid="clrnormal"
    activetextid="clrwarning"
    font="fntmenusm"
    droplist="menuarrsm"
    droplistoff="menuoffsm">
    </style menu>
  </style>
```

FILE: "FORM_ADVANCE.DAT"

Line 28: Change the field reference from "name" to "thingname".

Line 123: Change the field reference from "name" to "thingname".

Line 222: Replace the code for shrinking the size to the new code shown below.

```
perform portal[name].sizetofit[36]
perform portal[name].centervert
```

Line 238: Replace the "label" element with the new behavior shown below.

```
<label>
   <labeltext><![CDATA[
    ~use any domain term specified, else default to "domain"
    @text = parent.tagnames[DomainTerm.?]
    if (empty(@text) <> 0) then
     @text = "Domain"
     endif
    @text &= ":"
    ]]></labeltext>
   </label>
```

Line 271: Insert the new template definition shown below.

```
 <template
   id="advNotate"
   name="Notation Specification"
   compset="AdvDetails">

   <portal
```

```
    id="lblnotate"
    style="lblStatic">
    <label
      text="Notation:">
      </label>
    </portal>

  <portal
    id="notation"
    style="editNormal">
    <edit
      field="advUser"
      maxlength="50">
      </edit>
    </portal>

  <position><![CDATA[
    ~set up our width and height
    height = portal[notation].height
    if (issizing <> 0) then
      done
      endif

    ~center everything vertically
    perform portal[lblnotate].centervert
    perform portal[notation].centervert

    ~position the label on the left with the edit portal next to it
    portal[lblnotate].left = 0
    perform portal[notation].alignrel[ltor,lblnotate,7]
    portal[notation].width = width - portal[notation].left
    ]]></position>
  </template>
```

Line 309: Replace the script code for setting the line height to the code shown below.

```
portal[notes].lineheight = 4
```

Line 320: Insert the template reference shown below.

```
<templateref template="advNotate" thing="advDetails" taborder="20"/>
```

Line 345: Insert the code below to position the notation template

```
~position the notation template in the same place
template[advNotate].left = portal[advNew].left
template[advNotate].top = portal[advNew].top
template[advNotate].width = portal[advNew].width
```

Line 347: Insert the code below to set the notation template to non-visible.

```
template[advNotate].visible = 0
```

Line 356: Insert the code below to show the notation template when appropriate.

```
elseif (container.parent.tagis[Advance.Notation] <> 0) then
  template[advNotate].visible = 1
```

FILE: "FORM_DASHBOARD.DAT"

Line 121: Replace the Label script code with the new code shown below.

```
@text = "{size 30}PP: {size 36}" & hero.child[trkPower].field[trkUser].text
```

Line 189: Added the "isbuiltin" attribute to the element with a value of "yes".

FILE: "FORM_STATIC.DAT"

Line 59: Insert the new portal definition below.

```
<portal
  id="master"
  style="actMaster"
  tiptext="Click this button to activate this ally's Master.">
  <action
    action="master">
    </action>
  </portal>
```

Line 76 and 87: Put the contents of the Position script into a "<![CDATA[...]]>" block.

Line 77: Insert the code below at the start of the Position script.

```
~only show the master button if the actor is a minion
 portal[master].visible = hero.isminion  if (portal[master].visible <> 0) then
  perform portal[label].alignrel[ltor,master,8]
  endif
```

FILE: "FORM_TACCON.DAT"

Line 179: Insert the "isbuiltin" attribute with a value of "yes".

Line 189: Insert the "isbuiltin" attribute with a value of "yes".

Line 199: Insert the "isbuiltin" attribute with a value of "yes".

Line 203: Insert the new portal definition shown below.

```
<portal
  id="dead"
  style="imgNormal"
  tiptext="This character is dead or otherwise out of combat.">
  <image_literal
    image="tactical_dead.bmp"
    isbuiltin="yes"
    istransparent="yes">
    </image_literal>
  </portal>
```

Line 216: Change the style assigned to the portal to "lblSmlLeft".

Line 225: Eliminate the left-alignment logic from the Label script as shown below.

```
~squeeze inter-line spacing a bit
 @text = "{leading -2}" & @text
```

Line 254: Replace the "damage" and "status" portals with the new portal definitions shown below.

```
<portal
  id="status1"
  style="lblSmlLeft">
  <label
    ismultiline="yes">
    <labeltext><![CDATA[
      ~start with the power points status
      @text = "{size 30}PP: {size 36}" & #traituser[trPowerPts] & " / " & #trait[trPowerPts]

     ~add the defense rating
      @text &= "{horz 12}{size 30}Def: {size 36}" & #trait[trDefense]
     ]]></labeltext>
   </label>
  </portal>

<portal
  id="status2"
  style="lblSmlLeft">
```

```
  <label
   ismultiline="yes">
   <labeltext><![CDATA[
    @text = "{size 30}HP: {size 36}" & field[acHPSumm].text
    ]]></labeltext>
  </label>
 </portal>
```

Line 451: Renamed the portal from "column1" to "traits".

Line 461: Change the "foreach" statement to reference the "DashTacCon.Traits" tag expression.

Line 517: Append "sortas _NameSeq_" to the end of the "foreach" statement.

Line 544: Replace the "foreach" statement with the new statement shown below.

```
foreach pick in hero where "(Adjustment.? | Helper.Activated) &  !InPlay.Permanent" sortas
  _NameSeq_
  if (ismore <> 0) then
    @text &= "; "
    endif
  if (eachpick.tagis[component.Adjustment] <> 0) then
    @text &= eachpick.field[adjShort].text
  elseif (eachpick.tagis[component.shortname] <> 0) then
    @text &= eachpick.field[shortname].text
  else
    @text &= eachpick.field[name].text
    endif
  ismore = 1
  nexteach
```

Line 669-681: Replace the script code that controls visibility and positioning with the new logic shown below.

```
~position the "dead" indicator in the same location
 portal[dead].left = leftedge
 perform portal[dead].centervert

~hide all of the indicators and we'll pick one to show below (or none)
 portal[dead].visible = 0
 portal[acted].visible = 0
 portal[noncombat].visible = 0
 portal[never].visible = 0
~if we're in combat, handle things appropriately
 if (state.iscombat <> 0) then

  ~determine which of the above four indicators is actually visible
   if (hero.tagis[Hero.Dead] <> 0) then
     portal[dead].visible = 1
   else
     portal[never].visible = hero.tagis[combat.never]
     portal[acted].visible = hero.tagis[combat.acted]
     portal[noncombat].visible = hero.tagis[combat.noncombat]
     endif

  ~adjust our left edge rightward past the indicators
   leftedge += portal[never].width + 4
   endif
```

Line 759-778: Replace the script code that references the renamed "damage" and "status" portals with the new code shown below.

```
~position the second status portal at the bottom of the region
 perform portal[status2].alignedge[bottom,-margin - 2]
 portal[status2].left = portal[name].left + 3

~position the first status portal in the region above the second status portal
 perform portal[status1].alignrel[btot,status2,-1]
 portal[status1].left = portal[status2].left

~if the status now overlaps the name, shift the status portals downward a
~little bit to make additional space
```

```
if (portal[status2].top < portal[name].bottom) then
  var adjust as number
  adjust = portal[name].bottom - portal[status2].top
  portal[status2].top += adjust
  adjust -= 1
  if (adjust > 1) then
    adjust -= 1
    endif
  portal[status1].top += adjust
  endif
```

Line 851: Replace the code manipulating the "column1" portal with the following.

```
~position the column of reminder traits
 portal[traits].top = margin + 1
 portal[traits].left = leftedge
 portal[traits].lineheight = 3
```

Line 860: Replace the reference to the "column1" portal with the "traits" portal.

Line 868: Insert the new code below to size the summary appropriately.

```
perform portal[summary].sizetofit[28]
```

Line 870-871: Replace the references to the "column1" portal with the "traits" portal.

FILE: "PROCEDURES.DAT"

Line 780: Replace the "foreach" statement with the new logic shown below.

```
foreach pick in hero where "(Adjustment.? | Helper.Activated) &  !InPlay.Permanent"
  final &= "{br}"
  if (eachpick.tagis[Adjustment.?] <> 0) then
    final &= eachpick.field[adjName].text
  else
    final &= eachpick.field[name].text
    endif
  nexteach
```

FILE: "SHEET_STANDARD1.DAT"

Line 485: Replace the "dots" portal with the new definition below.

```
<portal
  id="dots"
  style="outDots">
  <output_dots>
    </output_dots>
  </portal>
```

Line 627: Insert the new logic below into the Position script.

```
~limit our portal height to a single line of output
portal[details].lineheight = 1
```

Line 661: Replace the "dots" portal with the new definition below.

```
<portal
  id="dots"
  style="outDots">
  <output_dots>
    </output_dots>
  </portal>
```

Line 670: Replace the entire Position script with the new logic shown below.

```
~our height is the height of the tallest portal
 height = portal[name].height
 if (issizing <> 0) then
   done
   endif

~position the value at the right edge
perform portal[value].alignedge[right,0]

~size the name to fit the available space
  portal[name].width = portal[value].left – 10
  perform portal[name].sizetofit[40]
  perform portal[name].autoheight

 ~the dots should span the region between the name and the value
  perform portal[dots].alignrel[ltor,name,5]
  portal[dots].width = portal[value].left - 5 - portal[dots].left

 ~center all portals vertically
  perform portal[name].centervert
  perform portal[value].centervert
  perform portal[dots].centervert
```

Line 711: Replace the entire Label script with the new logic shown below.

```
 if (stackable = 0) then
    @text = ""
 elseif (field[stackQty].value = 1) then
    @text = ""
 else
    @text = field[stackQty].text & "x"
    endif
```

Line 731: Replace the entire Position script with the new logic shown below.

```
 ~our height is the height of the tallest portal
  height = portal[name].height
  if (issizing <> 0) then
    done
    endif

 ~assign a fixed width to the value and position the name to the right
  portal[value].width = 100
  perform portal[name].alignrel[ltor,value,20]

 ~size the name to fit the available space
  portal[name].width = width - portal[name].left
  perform portal[name].sizetofit[36]
  perform portal[name].autoheight

 ~center all portals vertically
  perform portal[value].centervert
  perform portal[name].centervert
```

Lines 811-826: Replace the block of script code with the new logic shown below.

```
 ~align everything horizontally
  perform portal[badstr].alignrel[ltor,equipped,5]
  perform portal[name].alignrel[ltor,badstr,5]
 ~size the name to fit the available space
  portal[name].width = width - portal[name].left
  perform portal[name].sizetofit[36]
  perform portal[name].autoheight
 ~center all portals vertically
  perform portal[badstr].centervert
  perform portal[equipped].centervert
  perform portal[name].centervert
 ~shift the "equipped" bitmap downward a little bit; this is because it is a
  ~lone bitmap drawn via encoded text, and bitmaps are never drawn within the
  ~descender portion of the text, which causes it to appear higher than we want it
```

```
portal[equipped].top += 4
```

Line 867: Replace the "name" portal with the new element shown below.

```
<portal
  id="name"
  style="outNameMed">
  <output_label
    field="shortname">
    </output_label>
  </portal>
```

Line 912: Insert the new portal shown below.

```
<portal
  id="dots"
  style="outDots">
  <output_dots>
    </output_dots>
  </portal>
```

Lines 946-989: Replace the entire Position script with the new logic shown below.

```
~our height is based on the tallest portal within
 height = portal[name].height
 if (issizing <> 0) then
   done
   endif

~if the weapon satisfies the minimum strength requirement, hide the bitmap
 if (tagis[Helper.BadStrReq] = 0) then
   portal[badstr].visible = 0
   endif

~center all portals vertically
 perform portal[badstr].centervert
 perform portal[name].centervert
 perform portal[attack].centervert
 perform portal[damage].centervert
 perform portal[dots].centervert

~position the range with the same baseline as the rest of the text; since it
 ~uses a smaller font, it will have a smaller height, so centering it will have
 ~it appear to float up relative to the other text
 perform portal[range].alignrel[btob,name,0]

~establish suitable fixed widths for the various columns of data
 portal[damage].width = 120
 portal[attack].width = 70
 portal[range].width = 260

~position everything horizontally, leaving a margin on both sides appropriately
 portal[badstr].left = 5
 perform portal[damage].alignedge[right,-5]
 perform portal[name].alignrel[ltor,badstr,5]
 perform portal[attack].alignrel[rtol,damage,-10]
 perform portal[range].alignrel[rtol,attack,-10]

~if this is a ranged weapon, limit the name to the space up to the range details;
 ~otherwise, let the name extend over to the attack value
 var limit as number
 if (tagis[component.WeapRange] <> 0) then
   limit = portal[range].left
 else
   limit = portal[attack].left
   endif

~limit the name to the extent determined above
 if (portal[name].right > limit - 5) then
   portal[name].width = limit - portal[name].left – 5
```

```
  endif

~size the name to fit the available space
 perform portal[name].sizetofit[36]
 perform portal[name].autoheight
 perform portal[name].centervert

~extend the dots from the right of the name across to the value on the right
 if (portal[name].right > limit - 10) then
  portal[dots].visible = 0
 else
  perform portal[dots].alignrel[ltor,name,5]
  portal[dots].width = limit - 5 - portal[dots].left
  endif
```

Line 1039: Replace the entire Position script with the new logic shown below.

```
~our height is the vertical extent of our portals
 height = portal[name].height
 if (issizing <> 0) then
  done
  endif

~size the name to fit the available space
 portal[name].width = width
 perform portal[name].sizetofit[36]
 perform portal[name].autoheight
 perform portal[name].centervert
```

Line 1173: Replace the code for calculating the validation report height with the new logic below.

```
perform portal[validate].autoheight
 if (portal[validate].height > portal[validate].fontheight * maxlines) then
  portal[validate].lineheight = maxlines
  endif
```

Numerous Places: Changes references to the "global" target reference over to "scenevalue". The list of locations is given below.

- Line 1208
- Line 1229
- Line 1315
- Line 1349
- Line 1442
- Line 1446
- Line 1464
- Line 1479
- Line 1480
- Line 1490

FILE: "SHEET_STANDARD2.DAT"

Line 70: Changes reference to the "global" target reference over to "scenevalue".

FILE: "SUMM_ARMORY.DAT"

Line 76: Replace the "name" portal with the new element shown below.

```
<portal
  id="name"
  style="lblSummary">
  <label>
    <labeltext><![CDATA[
      if (field[grIsEquip].value <> 0) then
        @text = "{b}{i}"
        endif
      @text &= field[name].text
      ]]></labeltext>
```

```
  </label>
 <mouseinfo/>
 </portal>
```

Line 118: Replace the "name" portal with the new element shown below.

```
<portal
  id="name"
  style="lblSummary">
  <label>
   <labeltext><![CDATA[
    if (field[grIsEquip].value <> 0) then
      @text = "{b}{i}"
      endif
    @text &= field[name].text
    ]]></labeltext>
  </label>
  <mouseinfo/>
  </portal>
```

FILE: "TAB_ARMORY.DAT"

Line 86: Change the component referenced from "WeapRange" to "Gear".

Line 265: Insert the "isbuiltin" attribute with a value of "yes".

Line 277: Insert the "isbuiltin" attribute with a value of "yes".

Line 319: Insert the "isbuiltin" attribute with a value of "yes".

Line 519: Insert the "isbuiltin" attribute with a value of "yes".

Line 531: Insert the "isbuiltin" attribute with a value of "yes".

FILE: "TAB_GEAR.DAT"

Line 122: Change the style from "lblNormal" to "lblLeft".

Line 143: Insert the "isbuiltin" attribute with a value of "yes".

Line 155: Insert the "isbuiltin" attribute with a value of "yes".

Line 198: Replace the entire Position script with the new logic shown below.

```
~set up our height based on our tallest portal
 height = portal[info].height

~if this is a "sizing" calculation, we're done
 if (issizing <> 0) then
   done
   endif

~determine whether the container and heldby indicators should be visible
 portal[container].visible = tagis[thing.holder?]
 portal[heldby].visible = isgearheld

~center the portals vertically
 perform portal[info].centervert
 perform portal[name].centervert
 perform portal[username].centervert
 perform portal[gearmanage].centervert
 perform portal[delete].centervert
 perform portal[container].centervert
 perform portal[heldby].centervert

~position the delete portal on the far right
 perform portal[delete].alignedge[right,0]

~position the info portal to the left of the delete button
 perform portal[info].alignrel[rtol,delete,-8]

~position the gear portal to the left of the info button
```

```
 perform portal[gearmanage].alignrel[rtol,info,-8]

~calculate the space to reserve for the various indicators
 var reserve as number
 if (portal[heldby].visible <> 0) then
   reserve += portal[heldby].width + 2
   endif
 if (portal[container].visible <> 0) then
   reserve += portal[container].width + 2
   endif
 if (portal[heldby].visible + portal[container].visible <> 0) then
   reserve += 3
   endif

~position the name on the left and let it use all available space
 var limit as number
 limit = portal[gearmanage].left - 8 – reserve
 portal[name].left = 0
 portal[name].width = minimum(portal[name].width,limit)

~if this is a "custom" gear pick, show an edit portal instead of the name
 var nextleft as number
 if (tagis[Equipment.CustomGear] <> 0) then
   portal[name].visible = 0
   portal[username].left = portal[name].left
   portal[username].width = minimum(200,limit)
   nextleft = portal[username].right
 else
   portal[username].visible = 0
   nextleft = portal[name].right
   endif
 nextleft += 5

~show the 'container' icon to the right of the name (if visible)
 if (portal[container].visible <> 0) then
   portal[container].left = nextleft
   nextleft = portal[container].right + 2
   endif

~show the 'held by' icon to the right of the container icon (if visible)
 if (portal[heldby].visible <> 0) then
   portal[heldby].left = nextleft
   endif

~if the gear can't be deleted (i.e. it's been auto-added instead of user-added,
 ~set the style to indicate that behavior to the user
 if (candelete = 0) then
   perform portal[name].setstyle[lblAuto]
   endif
```

FILE: "TAB_JOURNAL.DAT"

Line 98: Replace the line of code that references the usage pool to the new line shown below.

```
@text &= "{horz 40} Total XP: " & #resmax[resXP]
```

FILE: "THING_ABILITIES.DAT"

Line 23: Insert the following material within the "abSample" ability.

```
<!-- If checkbox selection is needed, make sure the compset includes "UserSelect"
      component and define this field appropriately.
 <fieldval field="usrChkText" value="Menu1"/>
 -->

<!-- If thing-based menu selection is needed, make sure the compset includes
      "UserSelect" component and define these fields and tags as appropriate.
 <fieldval field="usrLabel1" value="Menu1"/>
 <fieldval field="usrCandid1" value="component.Attribute"/>
 <fieldval field="usrLabel2" value="Menu2"/>
 <fieldval field="usrCandid2" value="component.Skill"/>
```

```
 <tag group="ChooseSrc1" tag="Hero"/>  <tag group="ChooseSrc2" tag="Thing"/>
 -->

<!-- If array-based menu selection is needed, make sure the compset includes
      "UserSelect" component and define these fields as appropriate.
 <fieldval field="usrLabelAr" value="Menu1"/>
 <arrayval field="usrArray" index="0" value="Choice #1"/>
 <arrayval field="usrArray" index="1" value="Choice #2"/>
 <arrayval field="usrArray" index="2" value="Choice #3"/>
 -->
```

FILE: "THING_ARMORY.DAT"

Line 18: Insert the following thing definition.

```
<!-- Natural armor is automatically equipped -->
 <thing
   id="armNatural"
   name="Natural Armor"
   compset="Armor"
   description="Description goes here"
   isunique="yes"    holdable="no">
   <fieldval field="defDefense" value="1"/>
   <tag group="Equipment" tag="Natural"/>
   <tag group="Equipment" tag="AutoEquip"/>
   </thing>
```

FILE: "THING_ATTRIBUTES.DAT"

Line 18: Insert the following field value assignment.

```
<fieldval field="trtAbbrev" value="Sam"/>
```

Line 31: Insert the following field value assignment.

```
<fieldval field="trtAbbrev" value="Str"/>
```

FILE: "THING_SKILLS.DAT"

Line 18: Insert the following field value assignment.

```
<fieldval field="trtAbbrev" value="Sam"/>
```

Line 37: Insert the following field value assignment.

```
<fieldval field="trtAbbrev" value="Mel"/>
```

Line 50: Insert the following field value assignment.

```
<fieldval field="trtAbbrev" value="Sht"/>
```

FILE: "THING_TRAITS.DAT"

Line 38: Insert the following field value assignment.

```
<fieldval field="trtAbbrev" value="Hlth"/>
```

Line 60: Change the "Column1" tag reference to "Traits".

Line 77: Insert the following field value assignment.

```
<fieldval field="trtAbbrev" value="Def"/>
```

Line 96: Insert the following field value assignment.

```
<fieldval field="trtAbbrev" value="Powr"/>
```

Line 117: Change the "Column1" tag reference to "Traits".

Line 147: Change the "Column1" tag reference to "Traits".

FILE: "VISUAL.DAT"

Line 130: Insert the following code at the end of the Position script.

```
~center the name if requested
 if (tagis[SimpleItem.CenterName] <> 0) then
   perform portal[name].centerhorz
   endif
~if this is an auto-added pick, change its font to indicate that fact
 if (ispick +  !candelete >= 2) then
   perform portal[name].setstyle[lblAuto]
   endif
```

Line 213: Insert the new template definition below.

```
<!-- UserSelect template
      Similar to SimpleItem, except that this template is only suitable for
      showing picks and the items can employ various mechanisms for customizing
      the contents of the pick in some way. This template can be used or readily
      adapted when you integrate the "UserSelect" component into a component set
      and want to let the user customize the pick contents. For more details,
      please refer to the "UserSelect" component.
 -->
 <template
   id="UserSelect"
   name="User Selection"
   compset="UserSelect"
   marginhorz="3"
   marginvert="2">

  <portal
    id="name"
    style="lblNormal"
    showinvalid="yes">
    <label
      field="thingname">
      </label>
    </portal>

  <portal
    id="lblmenu1"
    style="lblSecond">
    <label
      field="usrLabel1">
      </label>
    </portal>

  <portal
    id="lblmenu2"
    style="lblSecond">
    <label
      field="usrLabel2">
      </label>
    </portal>

  <portal
    id="menu1"
    style="menuNormal">
    <menu_things
      field="usrChosen1"
      component="none"
```

```
      maxvisible="10"
      usepicksfield="usrSource1"
      candidatefield="usrCandid1">
      </menu_things>
    </portal>

  <portal
    id="menu2"
    style="menuNormal">
    <menu_things
      field="usrChosen2"
      component="none"
      maxvisible="10"
      usepicksfield="usrSource2"
      candidatefield="usrCandid2">
      </menu_things>
    </portal>

  <portal
    id="lblmenuar"
    style="lblSecond">
    <label
      field="usrLabelAr">
      </label>
    </portal>
<portal
  id="menuarray"
  style="menuNormal">
  <menu_array
    field="usrSelect"
    array="usrArray"
    maxvisible="10">
    </menu_array>
  </portal>

<portal
  id="checkbox"
  style="chkNormal">
  <checkbox
    field="usrIsCheck"
    dynamicfield="usrChkText">
    </checkbox>
  </portal>

<portal
  id="info"
  style="actInfo">
  <action
    action="info">
    </action>
  <mouseinfo/>
  </portal>

<portal
  id="delete"
  style="actDelete"
  tiptext="Click to delete this item">
  <action
    action="delete">
    </action>
  </portal>

<position><![CDATA[
  ~set up our height based on our tallest portal
  height = portal[info].height

 ~if this is a "sizing" calculation, we're done
  if (issizing <> 0) then
    done
    endif

 ~position our tallest portal at the top
  portal[info].top = 0

 ~center the other portals vertically
```

```
 perform portal[name].centervert
 perform portal[delete].centervert
 perform portal[lblmenu1].centervert
 perform portal[menu1].centervert
 perform portal[lblmenu2].centervert
 perform portal[menu2].centervert
 perform portal[lblmenuar].centervert
 perform portal[menuarray].centervert
perform portal[checkbox].centervert


~determine whether our portals are visible; we only show them if requested
 ~Note: Remember that a non-empty tagexpr field indicates menu selection is used.
 if (field[usrCandid1].isempty <> 0) then
   portal[lblmenu1].visible = 0
   portal[menu1].visible = 0
 elseif (field[usrLabel1].isempty <> 0) then
   portal[lblmenu1].visible = 0
   endif
 if (field[usrCandid2].isempty <> 0) then
   portal[lblmenu2].visible = 0
   portal[menu2].visible = 0
 elseif (field[usrLabel2].isempty <> 0) then
   portal[lblmenu2].visible = 0
   endif
 if (empty(field[usrArray].arraytext[0]) <> 0) then
   portal[lblmenuar].visible = 0
   portal[menuarray].visible = 0
 elseif (field[usrLabelAr].isempty <> 0) then
   portal[lblmenuar].visible = 0
   endif
 if (field[usrChkText].isempty <> 0) then
  portal[checkbox].visible = 0
   endif

~position the delete portal on the far right and the info portal next to it
 perform portal[delete].alignedge[right,0]
 perform portal[info].alignrel[rtol,delete,-8]

~determine our effective right edge, allowing for the buttons above
 var edge as number
 edge = portal[info].left - 10

~setup the default portal width and gap to be used between and around portals
 var defwidth as number
 var gap as number
 defwidth = 100
 gap = 10

~determine the minimum amount of space we need to reserve for our portals
 var reserve as number
 if (portal[checkbox].visible <> 0) then
   reserve = defwidth     elseif (portal[menuarray].visible <> 0) then
   reserve = portal[lblmenuar].width * portal[lblmenuar].visible
   reserve += defwidth + gap
 elseif (portal[menu1].visible <> 0) then
   reserve = portal[lblmenu1].width * portal[lblmenu1].visible
   reserve += defwidth + gap
   reserve += portal[lblmenu2].width * portal[menu2].visible
   reserve += (defwidth + gap) * portal[menu2].visible
   endif

~position the name on the left, reserving our minimum space for any portals
 var x as number
 portal[name].left = 0
 portal[name].width = minimum(portal[name].width,edge - portal[name].left - reserve)
 x = portal[name].right + gap

~setup the maximum width for our some portals, regardless of space available
 var maxwidth as number
 maxwidth = 150

~if we have a checkbox, size and position it appropriately
 if (portal[checkbox].visible <> 0) then
```

```
    portal[checkbox].left = x

  ~if we have an array-based menu, size and position it appropriately
   elseif (portal[menuarray].visible <> 0) then
    if (portal[lblmenuar].visible <> 0) then
      portal[lblmenuar].left = x
      x = portal[lblmenuar].right + 4
      endif
    portal[menuarray].left = x
    portal[menuarray].width = maxwidth

  ~if we have one thing-based menu, size and position it appropriately
   elseif (portal[menu1].visible + portal[menu2].visible = 1) then
    if (portal[lblmenu1].visible <> 0) then
      portal[lblmenu1].left = x
      x = portal[lblmenu1].right + 4
      endif
    portal[menu1].left = x
    portal[menu1].width = minimum(edge - portal[menu1].left,maxwidth)

  ~if we have two thing-based menus, size and position them appropriately
   elseif (portal[menu1].visible <> 0) then
    if (portal[lblmenu1].visible <> 0) then
      portal[lblmenu1].left = x
      x = portal[lblmenu1].right + 4
      endif
    portal[menu1].left = x
    var extra as number
    extra = (portal[lblmenu2].width + 4) * portal[lblmenu2].visible
    var actual as number
    actual = (edge - portal[menu1].left - extra - gap) / 2
    portal[menu1].width = minimum(actual,maxwidth)
    portal[menu2].width = portal[menu1].width
    x = portal[menu1].right + gap
    if (portal[lblmenu2].visible <> 0) then
      portal[lblmenu2].left = x
      x = portal[lblmenu2].right + 4
      endif
    portal[menu2].left = x
     endif

   ~if a menu is visible, make sure it has a selection
    if (portal[menu1].visible <> 0) then
      if (field[usrChosen1].ischosen = 0) then
        perform portal[menu1].setstyle[menuError]
        endif
      endif
    if (portal[menu2].visible <> 0) then
      if (field[usrChosen2].ischosen = 0) then
        perform portal[menu2].setstyle[menuError]
        endif
      endif
    if (portal[menuarray].visible <> 0) then
      if (field[usrSelect].isempty <> 0) then
        perform portal[menuarray].setstyle[menuError]
        endif
      endif
   ]]></position>

  </template>
```

SKELETON FILE CHANGES V3.2

FILE: "TAGS.1ST"

Line 39: Added definition of the "Helper.NoMinimum" tag for use with Resources.

FILE: "EQUIPMENT.STR"

Line 68: Corrected the calculation of the "lot cost" for a piece of gear.

Line 767: When creating ammunition and we have a transaction pick, the initial user quantity must be setup based on the actual quantity purchased.

FILE: "TRAITS.STR"
Line 214: Added the "Activated" identity tag group for shared use with adjustments.

Line 240: If an ability is activated, the corresponding "Activated" identity tag is now forwarded to the actor.

FILE: "MISCELLANEOUS.STR"
Line 182: Added new Eval Rule script that reports a validation warning for any resource that is not fully spent, except if that resource is assigned the "Helper.NoMinimum" tag.

FILE: "THING_MISCELLANEOUS.DAT"
Line 81: Assigned "Helper.NoMinimum" tag to disable reporting a validation warning if XP are not spent.

FILE: "FORM_STATIC.DAT"
Line 81: Eliminated the "maxlength" attribute that now results in a compilation error.

FILE: "FORM_CONFIG.DAT"
Line 92: Eliminated the "maxlength" attribute that now results in a compilation error.

FILE: "FORM_MANIPULATE.DAT"
Line 49: Eliminated the "maxlength" attribute that now results in a compilation error.

FILE: "SHEET_STANDARD2.DAT"
Line 85: The right-side column of output must be properly positioned on the right.

FILE: "EDITOR.DAT"
Line 17: Added "prefix" attribute for use by the Editor.

Line 42: Added "prefix" attribute for use by the Editor.

Line 72: Added "prefix" attribute for use by the Editor.

Line 119: Added "prefix" attribute for use by the Editor.

Line 129: Added "prefix" attribute for use by the Editor.

Line 149: Added "prefix" attribute for use by the Editor.

Line 194: Added "prefix" attribute for use by the Editor.

Line 254: Added "prefix" attribute for use by the Editor.

Line 279: Added "prefix" attribute for use by the Editor.

Line 156: Added "Ammunition" edit thing to allow users to enter ammunition via the Editor.

Line 137: Added "Lot Cost" and "Weight" fields for use by the Editor.

Line 207: Added "Weight" field for use by the Editor.

Line 272: Added "Weight" field for use by the Editor.

Line 317: Added "Weight" field for use by the Editor.

Line 347: Added "Weight" field for use by the Editor.

FILE: "SYSTEM_RESOURCES.AUG"
Line 268: Added definition of the "tabwarning" color resource to allow authors to easily override the resource if they wish.

Line 626: Added definition of the "edtbackov" color resource to enable overrides.

Line 1131: Added definition of the "progtext" color resource to enable overrides.

Line 1315: Added definition of the "cnflictext" color resource to enable overrides.

Line 1315: Added definition of the "buygameup" and "buygamedn" bitmap resources to enable overrides.

XML CHARACTER ENCODING SET

HL assumes all XML documents utilize the "ISO-8859-1" character set (also called Latin-1), with a number of exceptions specific to the Windows platform. The list of exceptions is detailed in the table below.

128	undefined	144	undefined
129	undefined	145	‘
130	,	146	’
131	*f*	147	“
132	„	148	”
133	…	149	•
134	†	150	–
135	‡	151	—
136	ˆ	152	˜
137	‰	153	™
138	Š	154	š
139	‹	155	›
140	Œ	156	œ
141	undefined	157	undefined
142	undefined	158	undefined
143	undefined	159	Ÿ

The identity element at the top of all XML files should specify an encoding of "ISO-8859-1" for completeness. If no encoding is given, ISO-8859-1 is assumed. An example is given below:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
```

NOTE! There is an unofficial XML encoding named "Windows-1252" that properly reflects the Windows ANSI character set and is often used. However, various XML parsers do not recognize this encoding set due to its unofficial nature. In the interest of maximum compatibility, the modified Latin-1 set is used instead.

XML ATTRIBUTES IN DATA FILES

All of the information contained within data files is managed through XML elements, attributes, and PCDATA. Each XML element typically has a number of attributes which dictate the characteristics of that element. Some attributes will be required and some will be optional. When an attribute is optional, it will be clearly designated as such, along with the default value. Unless specified as optional, all attributes are required.

The XML specification allows attributes to represent virtually anything with a text value. However, the Kit differentiates between various types of attributes, each of which has various rules associated with the values it can be assigned. Within the Kit Reference, each attribute's type will be specified in its description. The following types of attributes are utilized, with the indicated constraints being imposed.

- **Text:** Text attributes must always consist of standard ASCII text characters. Unless specified otherwise in the description of the attribute, text attributes have no additional constraints other than those dictated by the use of the attribute's contents. For example, a text attribute that specifies a file name needs to comply with the naming rules for files, else attempts to use the file name will fail. Any special constraints are indicated within the attribute description.
- **Integer:** Integer attributes must specify an integer value. In general, integer attributes will represent positive numbers, although it is valid to specify a negative value when appropriate.
- **Float:** Float attributes must specify a floating point value (e.g. 3.14159). In general, only positive values are used, although negative values are sometimes allowed. The attribute description will specify any restrictions.
- **Boolean:** Boolean attributes must specify a value of either “yes” or “no”. The name of the Boolean attribute will typically give a clear indication of the meaning of “yes” or “no”. For example, tag groups have a “visible” attribute, where the meanings of “yes” and “no” are clear.
- **Id:** Ids represent the unique id of an object. The rules for Unique Ids are outlined separately.

- **Set:** Set-based attributes must specify one of an explicit set of values. Boolean attributes are an example of a set-based attribute, wherein the value must be specified as one of the two valid choices: "yes" and "no". Set-based attributes will specify the valid values and their meanings.

NOTE! Many attributes of type "text" will have an appropriate maximum length specified in their description. It is perfectly valid to specify a value that exceeds the indicated maximum within an XML file, since XML imposes no constraints. However, if you exceed the designated maximum length for a text attribute, HL will either report an error or simply truncate the text to the maximum length allowed (depending on the context). This can potentially yield unintended results, so be careful when specifying text attributes. Some text attributes have no maximum length, in which case HL will utilize whatever you specify for the attribute.

SPECIFYING PCDATA IN DATA FILES

Numerous elements throughout the various XML files utilize the PCDATA block of the element to hold important information. PCDATA blocks, in conjunction with the "<[CDATA[" section, make it possible to easily enter lengthy text that does not need to worry about the various XML restrictions on certain characters used in the text.

To simplify the documentation, the use of PCDATA within an element will not be handled separately. Instead, any element that uses PCDATA will simply have an attribute listed with the name "PCDATA". The description for this attribute will apply to the use of the PCDATA block within the element.

One characteristic of PCDATA is that it can optionally preserve or collapse all use of whitespace characters within the block. Whitespace characters include spaces, tabs, and newlines. When whitespace is collapsed, text that runs across multiple lines is merged together into a single flow, which is ideal when you're writing paragraphs of information, such as release notes. However, there are many times when it is critical that whitespace be preserved, such as with scripts that require each statement to be on a separate line.

When a PCDATA block is used as a script, the attribute type will be designated as "Script" to distinguish it from the normal "Text" type. By default, whitespace will be preserved for scripts and collapsed for all other situations. Any exceptions to this behavior will be clearly specified within the description for the PCDATA attribute.

OPTIONAL ATTRIBUTES IN DATA FILES

Information that is defined through XML attributes and/or PCDATA blocks is always required by default. The exception is an entry that is specifically designated as optional. If an entry is optional, it will be clearly designated as such within its description, with the text "(Optional)" appearing before the type is given. An optional attribute will always specify the default value that is used if the attribute is left blank. Optional PCDATA blocks are always treated as empty by default.

LEVERAGING TAGS VIA TAG EXPRESSIONS

The Kit utilizes tags as a fundamental building block upon which a substantial number of mechanisms are based. While scripts and field values are critical components as well, you'll often rely on tags to determine when and how to process those scripts and fields. This section provides in-depth details on how to leverage tags in your design and how tags are utilized through tag expressions.

In the topics below, we focus on the various ways in which tags can be utilized. The syntax details are quite similar for each, since all are built upon the same fundamental elements. There are four basic layers of use that progressively build upon each other.

Simply click on the topics below to learn more about it.

IMPORTANT! This section assumes you are already familiar with the basics of tags and tag expressions. If you have not already done so, please review the section Data Manipulation Basics before proceeding with this section.

TAG TEMPLATES
Tag templates provide a simple mechanism to determine if an entity contains a specific tag. Templates also provide wildcard comparisons.

TAG TERMS
Tag terms build upon tag templates to determine if entities comply with a specific criteria, whether it be the presence of a tag or an arithmetic comparison against a tag.

TAG EXPRESSIONS
Tag expressions utilize Boolean logic to combine tag terms into complex expressions.

ARITHMETIC EXPRESSIONS
Arithmetic expressions enable the creation of calculations based upon the tags assigned to objects in the portfolio.

TAG TEMPLATES
All things are assigned an assortment of tags that reflect the various characteristics of those things. Tags can be utilized by the data file author to identify things that must adhere to rules or should be subject to specific types of processing. You can assign tags to picks and

containers dynamically during evaluation processing to change their nature and behavior. Tags can even be utilized by the end- user to filter things in arbitrary combinations. So now the question is how to identify the things that have the specific tags you are seeking. The basic mechanism is called a tag template.

BASIC USAGE AND SYNTAX

Tag templates represent a kind of "query" that compares an object against a specific tag, or possibly a group of similar tags using a wildcard. Either the object contains one or more tags that match the template or not. In general, a tag template returns the number of tags within the object that match the template. How that value is used depends on the context in which it is used.

Tag templates identify a specific tag group and a comparison string that can match zero or more tags within that group. The syntax for a tag template is "group.compare", where "group" is the unique id of a specific tag group and "compare" is a comparison string to match against individual tags within the group. The two values are always separated by a period. The comparison string always compares against the unique id assigned to tags.

Note that tag templates always utilize the unique ids of the tag and the group. This makes tag usage completely independent of that actual names used for tags and/or tag groups.

For example, a tag template to identify all objects of the color red would be defined as "color.red". The unique id of the tag group is given first ("color"), followed by the unique id of the tag itself ("red").

WILDCARDS

The simplest tag template specifies an explicit tag (e.g. "color.red"). However, the '?' character can be placed at the end of a comparison string to denote a wildcard. When a wildcard is used, the template will match all tags whose unique id matches the comparison string up to the '?'. For example, the tag template "color.bl?" would match both of the tags "black" and "blue" for the group "color", but it would not match the tag "red". The wildcard character can only be specified at the **end** of a comparison string. Any characters occurring after the '?' will trigger a compiler error as invalid syntax.

Some examples of tag templates are provided below to illustrate how they can be used:

- **color.red:** Matches only the tag "red" within the group "color".
- **color.bl?:** Matches any tag that starts with the letters "bl" within the group "color". For example, this template would match tags for the colors "black" and "blue", but it would not match "red". It would also match the tag "bl".
- **color.?:** Matches any tag within the group "color", provided the object has at least one "color" tag. If the object does not possess any tags from the group "color", the template will resolve to "false".

IMPORTANT CONSIDERATIONS

Keep the following issues in mind when dealing with tag templates.

- All tag ids are case-specific, and so are tag templates. Be careful to specify the correct case for all templates.
- The comparison string utilizes the unique id for tags. If there are a number of related tags with which you want to be able to utilize wildcards, consider assigning them unique ids that make this possible. For example, if you have a "bigorc" and a "smallorc" tag, you could rename them as "orcbig" and "orcsmall". Using wildcards, you could reference both tags via the comparison string "orc?". This naming technique allows you to reference the two "orc" tags either independently or collectively, as the situation dictates.
- The tag template "group.?" can be extremely useful in various situations. Table selection and validation rules are often specific to objects from a given category. By creating a tag group for the desired category and using the "group.?" syntax, you can instantly identify all objects from a particular category of interest.

TAG TERMS

Tag expressions are built upon individual tag terms. Within a tag expression, every term is evaluated against the tags assigned to an object. Terms can either be "simple" terms or "arithmetic comparison" terms. Every individual term yields a result of either "true" or "false", allowing the terms to be combined into more complex Boolean expressions.

SIMPLE TERMS

Simple terms are just that – simple. A simple term must be one of the following:

- **TRUE:** The literal value "TRUE" can be specified (case is important).
- **FALSE:** The literal value "FALSE" can be specified (case is important).
- **tag template:** A tag template can be specified, with the term returning a Boolean result of true or false. When a tag template is evaluated as a simple term, the result is true if the object contains one or more tags that match the tag template and false if no tags match the template.

ARITHMETIC COMPARISON TERMS

There are times when you will need to identify objects which satisfy criteria that don't simply yield a "match" or "no match" result. In these instances, the object is required to comply with numeric constraints associated with the tags assigned. For example, a rule might pertain only to spells with a level of 5 or more. In situations like this, it's not appropriate to test for all the possible values (e.g. for the proposed sample rule, you would not want to test separately for spells of levels 5, 6, 7, 8, etc.). To handle special relationships like this, tag terms can be defined using arithmetic comparisons.

Arithmetic comparison terms always take the traditional form "left oper right", where "oper" is one of the arithmetic comparison operators: '<', '<=', '>', '>=', '=', or '<>' (the last one indicating "not equal"). Both "left" and "right" must evaluate to numeric values, with the comparison term yielding a "true" or "false" result based on the relationship between the values for "left" and "right". In this type of usage, the left and right sides of the comparison must be one of the following:

- **Integer:** A literal integer value can be specified (e.g. "1234"). Use of a literal integer value is restricted to the **right** side of an arithmetic comparison term only. An error occurs if a literal integer value is used on the left side. The integer value may not be negative.
- **count:template:** A standard tag template of the form "group.template", subscribing to the syntax set forth previously (e.g. "color.bl?"). The value generated is the number of tags that have been assigned to the object which match the template (wildcards are allowed per standard tag template syntax).
- **low:template:** All tags that match the specified template are identified, and an integer value is extracted from the end of each tag. The lowest (i.e. minimum) value of all tags is the value yielded. Integer value extraction is described below. Example: "low:cost.gold?".
- **high:template:** Identical to "low:template", except that the highest (i.e. maximum) value of all matching tags is yielded. Integer extraction is described below. Example: "high:cost.gold?".
- **val:template:** The first tag that matches the specified template is identified, and an integer value is extracted from that tag and returned. Integer value extraction is described below. Example: "val:cost.gold?". **NOTE!** If the template matches multiple tags, there is no guarantee which tag will be retrieved, so this form should only be used when you know there is a single tag matching the template assigned to the object. Using this form is much more efficient that the "low" and "high" forms, so make use of it whenever it is safe to do so.

NOTE! The prefix may optionally be omitted from a tag term within an arithmetic expression. If no prefix is specified, then the default prefix of "count" is assumed. Therefore, the term "color.blue" is equivalent to "count:color.blue" within an arithmetic comparison term, such as below.

```
(color.blue >= 1)
```

TAG VALUE EXTRACTION

Every tag can be converted to an integer value. Extracting an integer value from a tag utilizes the unique id of the tag and not its name. Starting at the end of the tag's unique id, all digits are extracted until a non-digit is encountered. Those digits are then converted to an integer value. For example, integer value extraction from the tag "blue1234" would yield the value "1234". Similarly, the tag "blue5x123" would yield the value "123".

If a tag does not end in any digits, then the tag will yield "no" value, which will be treated as a value of zero when an integer value is required for the tag. For example, the tag "blue567xyz" would yield no value, since only the trailing digits are used and all earlier digits are ignored.

NOTE! Depending on the situation, there may be a critical distinction between a tag with "no" value and a tag with a "zero" value.

Be sure to keep the following issues in mind when dealing with tag values:

- Negative values are never extracted from tags, since only digits are considered when extracting the value.
- Under some circumstances, a tag will have no value and be referenced as if it has a value. Tags with "no" value are ignored in those situations (e.g. within summations and averages), while a tag with a "zero" value is processed with that value.
- An object that does not have any tags matching the specified template and for which a value is referenced will never satisfy an arithmetic comparison under any circumstances. For example, the comparison "val:attack.? < 3" will always fail for any object that has no tag from the "attack" group. Similarly, the comparison "val:attack.? >= 3" will also fail for those same objects. If an object has no value, it can never be compliant with a value-based comparison.

FIELD VALUE TESTS

There are times when tracking a piece of information for an object really needs to be handled via a field value, yet you still need to test that state within tag expressions. One option would be to maintain both the field value and a corresponding tag that can be tested in a tag expression, but that approach quickly becomes difficult to maintain. To alleviate this, the Kit allows direct access to field values from within tag expressions.

You can reference field values using the syntax "fieldval:fieldid", where "fieldid" is replaced by the unique id of the field to access. Other than that, you can utilize field references just like any other arithmetic comparison term within the tag expression. For example, the tag expression "fieldval:strength < 10" would be satisfied if the object possesses a "strength" field and that field has a value of less than 10.

Only the fields for the appropriate object context can be referenced via tag expressions. Consequently, if the tag expression is being applied to a pick, only the fields defined for that pick can be referenced. If a field reference is used within a tag expression that either is not a pick/thing context or does not contain the specified field, a run-time error is reported to the user. If a tag expression will be used against things/picks that derive from different component sets and may not all possess a particular field, the author can include a test for the needed component within the tag expression to verify the presence of the field before accessing it. For example, if the field "foo" is defined within the "compid" component, the following syntax can be used to safely test the field value:

```
(component.compid & (fieldval:foo >= 1))
```

Be sure to keep the following important issues in mind when utilizing field values via tag expressions:

- Since only things and picks possess fields, access to field values will only work when the target object is a thing or a pick. There are times this might not be immediately obvious. For example, you might assume that field values can be referenced within a thing "condition" tag expression, except that the "condition" test is applied against the tags of the container, which means there is no thing/pick context to retrieve any fields from.
- In addition to a variety of standard tag expressions within a thing/pick context, access to field values can also be utilized from within most scripts. Any time that a pick target context is accessed from within a script, the "tagexpr[...]" target reference can safely utilize a field value references against the fields within that context.
- Do NOT use field value access as a crutch within tag expressions. Field value access from tag expressions is significantly more expensive to process than testing a tag, so use tags whenever possible and only use field values when absolutely necessary.

EXPLICIT SCOPE RESTRICTIONS

While not common, situations will arise where you'll want to test the tags of a thing/pick and the tags of the hero to determine whether a tag expression should be satisfied. For example, consider a situation where an assortment of things is only available if the actor belongs to a particular group of classes. Assuming those classes assign an identifiable tag to the actor, you could filter the things displayed based on the presence of appropriate tags on the things and the tag on the actor.

To support these situations, any tag template used within a tag expression can specify a prefix of "hero#" on the template. When this prefix is assigned, the tag template is compared directly against the tags of the containing hero context, even if a pick or a child container is the established context. For this to work, the prefix must appear immediately prior to the tag template, such as in the examples below.

```
group.tag & hero#group.tag
```

OR

```
val:hero#group.tag
```

NOTE! Unlike with scripts, you cannot transition through the hierarchy within the tag terms. You can test tags on the current context or directly on the hero. Those are the only options, and they will be all you need 99.9% of the time.

TAG EXPRESSIONS

Combinations of tags indicate meta-characteristics of an object. Tag expressions are the mechanism through which objects with specific combinations of attributes can be identified. Specifically, tag expressions are Boolean expressions that delineate a combination of tags that an object must possess. For example, a tag expression could easily be used to determine if an object is a "first-level wizard spell". Separate tag groups could be defined that identify the "class" of the object (fighter, wizard, etc.) and the "level" of the object (1st, 2nd, etc.). An additional tag group could be defined that identifies the "type" of the object (e.g. spell, weapon, skill, etc.). A tag expression could then be written to identify all objects that possess tags for all three characteristics: the "class.wizard" tag, the "level.1" tag, and the "type.spell" tag.

SYNTAX

Tag expressions are Boolean expressions comprised of tag terms. The tag expression evaluates each of the tag terms against a particular object to determine whether the object does or does not match the criteria. These terms are then combined using Boolean logic, where the expression specifies the relationships to use. Four Boolean operators are supported, and parentheses may be freely used to define appropriate groupings of sub-expressions. The operators supported are outlined below:

Opr	Example	Definition
&	"term1 & term2"	Logical "And". Evaluates the two terms on either side and returns "true" only if both terms are "true".

|	"term1 | term2"	Logical "Or". Evaluates the two terms on either side and returns "true" if either of the terms is "true".
^	"term1 ^ term2"	Logical "Xor". Evaluates the two terms on either side and returns "true" if exactly one of the terms is "true".
!	"!term1"	Logical "Not". Evaluates the following term and returns "true" if that term is "false". Otherwise, it returns "false".

EXAMPLES

The following are a couple of sample expressions utilizing parenthetical sub-expressions and combinations of the above operators.

```
(TRUE & FALSE) ^ (TRUE & TRUE) ^ (FALSE & FALSE)
```

The above example evaluates to "true". The first term evaluates to "false" and the second term evaluates to "true". The Xor of "true" and "false" (the first two terms) yields "true". The third term evaluates to "false". The Xor of "true" and "false" (the result of the first Xor and the third term) yields "true".

```
TRUE ^  !(FALSE | TRUE)
```

The above example evaluates to "true". The sub-expression "(FALSE | TRUE)" evaluates to "true". The negation of the sub-expression yields "false". The Xor of "true" and the negated sub-expression result of "false" yields "true".

ARITHMETIC EXPRESSIONS WITHIN TAG EXPRESSIONS

There will be times when you will need more than just a simple total of objects that match a set of criteria. For example, what if you need to calculate the combined total between two different groups of objects? Or the ratio between two groups of objects? In these situations, you will need to utilize an arithmetic expression to perform the calculation desired.

USAGE AND SYNTAX

Arithmetic expressions utilize the same basic syntax you learned back in elementary school. Expressions are formed out of numeric values that are combined using the standard arithmetic operators: '+', '–', '*', and '/'. An additional operator, %, can also be used that yields the modulus (i.e. remainder) after division (e.g. "5 % 3" would yield "2"). Arithmetic expressions can utilize parentheses to dictate how evaluation should be performed, with standard rules for arithmetic being used when no parentheses are specified (i.e. multiplication and division take precedence over addition and subtraction).

The numeric values within an arithmetic expression must be either a literal value of some sort (e.g. "3") or a standard tag term suitable for the context that yields a value. For example, within the context of a rule, tag terms can use the syntax "val:component.spell?". This same term could be used freely within an arithmetic expression that is utilized within a rule.

The following is an example tag expression that incorporates an arithmetic expression within it. In this example, the tag "color.red" must be defined, the tag "move.fast" must be defined, and the combined values of the "attack.?" and "defense.?" tags must be no more than 5.

```
color.red & (val:attack.? + val:defense.? <= 5) & move.fast
```

IMPORTANT CONSIDERATIONS

Be sure to keep the following issues in mind when dealing with arithmetic expressions.

- Arithmetic expressions may contain any number of terms within them. However, a maximum of 20 special terms may be utilized within a single arithmetic expression. A special term is any non-literal value (e.g. "val:attack.?"). This probably seems like an absurdly high number for any typical use for a role-playing game, but the limit needs to be specified for completeness. If you define an arithmetic expression that contains more than 20 special terms, the expression will fail to compile and be reported as containing a syntax error.

TAG EXPRESSION TYPES

The Kit leverages a variety of tag expressions for different purposes. The topics below provide a brief discussion of both the role and behavior of each different type of tag expression.

LIVE TAG EXPRESSION

The role of the Live tag expression is to determine whether a visual element should be considered "live". When a visual element is "live", it is fully operational within the interface. When non-live, a visual element is treated as if it does not exist, being omitted from display and not processed in any way. All attempts to manipulate a non-live visual element via scripts are simply ignored.

The Live tag expression is always applied against the container whose information is to be displayed within the visual element. This will typically be the currently active actor, although it will sometimes be a gizmo (when editing a gizmo) or a different actor (when displaying material in places like the Dashboard). The tag expression is compared against all of the tags possessed by the container at the conclusion of the evaluation cycle, so the effects of all dynamically assigned and deleted tags are included. If the tag expression is satisfied, the visual element is considered live and shown normally. Otherwise, the visual element is deemed non-live and hidden from display.

The Live tag expression is applied anew after every evaluation cycle. Consequently, it is possible to have visual elements dynamically appear and disappear in response to user actions. This can range from a simple portal changing state to an entire tab panel or summary panel transitioning its state.

CONTAINER TAG EXPRESSION

The role of the Container tag expression is to determine whether a structural element should be considered "live". When a structural element is "live", it is fully operational within over data hierarchy. When non-live, a structural element is treated as if it does not exist, so it is never shown to the user and no tasks associated with the element are processed in any way. All attempts to access or manipulate a non-live structure element via scripts will either be ignored or result in a run-time error.

The Container tag expression is always applied against the container of the structural element. In the case of a thing that has not yet been added to the portfolio, the tag expression is applied against the prospective container to which the thing will be added. The tag expression is compared against all of the tags possessed by the container, which includes the effects of all dynamically assigned and deleted tags. If the tag expression is satisfied, the structural element is considered live and behaves normally. Otherwise, the structural element is deemed non-live and its effects are ignored.

If the test occurs during the evaluation cycle, the test is applied against the current set of tags for the container. Consequently, it is critical that you pay attention to timing considerations and ensure that all the necessary tags are manipulated on the container prior to evaluating the tag expression.

The Container tag expression is applied whenever it is triggered. If that happens to be during the evaluation cycle, then it is applied during every evaluation cycle. This means it is possible to have structural elements dynamically change their live state in response to user actions. This in turn can cause structural elements to appear and disappear within the character, depending on how you setup your data files.

MATCH TAG EXPRESSION

The role of the Match tag expression is to determine whether a thing should be included in a set of elements for processing in some way. For example, an Eval script on a component will use a Match tag expression to identify the subsets of things to which the script should be applied. The implications of the Match tag expression are clear and simple, since the thing is either included for processing or not.

The Match tag expression is always applied against the thing that may be processed. The tag expression is compared against all of the tags possessed by the thing. Since tags possess a fixed set of tags, a given tag will always either satisfy a Match tag expression or not. If the tag expression is satisfied, the thing is included in the processing it is being considered for. Otherwise, the thing is omitted from processing.

LIST TAG EXPRESSION

The role of the List tag expression is to determine whether a pick or thing should be included in the list of items shown for a table- based portal. If the tag expression is satisfied, then the pick/thing is shown within the table. Otherwise, the pick/thing is omitted from the table.

The List tag expression is always applied against the pick/thing itself. The tag expression is compared against all of the tags possessed by the object. For picks, this includes the effects of all dynamically assigned and deleted tags. Since populating tables is performed after the evaluation cycle completes, the tags used for picks are the final set of tags when evaluation completes.

The List tag expression is applied anew after every evaluation cycle. Consequently, picks that dynamically change their state can appear and disappear within tables based on changes in how they satisfy the tag expression.

Any item that is added to the character via a given portal is **always** shown within that portal. This overrides the behavior of the List tag expression. Even if the object does not satisfy the List tag expression, it will be shown in the portal if it was previously added via the portal. This ensures that picks/things added via a portal are always made visible so that they can be readily deleted by the user.

CANDIDATE TAG EXPRESSION

The role of the Candidate tag expression is to determine whether a pick or thing should be included in a choose form. The choose form contains the list of picks/things from which the user can select when adding items to a portal (i.e. dynamic table, chooser, or thing- based menu). If the tag expression is satisfied, then the pick/thing is shown within the list of options and can be added by the user. Otherwise, the pick/thing is omitted from the available list.

The Candidate tag expression is always applied against the pick/thing itself. The tag expression is compared against all of the tags possessed by the object. For picks, this includes the effects of all dynamically assigned and deleted tags. The Candidate tag expression is

only invoked when the user triggers the portal to display the list of options to choose from. Consequently, the tags used for picks are the final set of tags after evaluation completes. This also means that picks can dynamically change their state and change whether they appear as valid choices within a given portal.

When a Candidate tag expression is used in conjunction with a List tag expression in a dynamic table, special behaviors can be leveraged. If you omit the Candidate tag expression entirely, the List tag expression is implicitly assumed, resulting in the user being able to select the same set of items that will be displayed. If you specify both tag expressions, the Candidate tag expression will normally supersede the List tag expression. However, if you specify the "inheritlist" attribute as "yes" on the Candidate tag expression, then the two tag expressions will be considered additive. Objects included in the selection list must then satisfy **both** the Candidate tag expression and the List tag expression. This allows you to avoid redundantly defining the logic of the List tag expression in both places while also ensuring that the items shown for selection correspond to the items that are intended to be shown in the table.

RESTRICTION TAG EXPRESSION

The role of the Restriction tag expression is to further limit the set of things or picks that are made available for selection through a portal. It is always used in conjunction with a Candidate tag expression and ensures that an item is only ever added to a specific table a single time. The Restriction tag expression is tested against all objects that are valid candidates for selection. If any of those objects also satisfies the Restriction tag expression **and** already exists within the table, they are precluded from being added to the table a second time and dropped from the selection list. This makes it possible to limit the user to only add an item once without having to designate the thing as unique.

The Restriction tag expression is always applied against the pick/thing itself. The tag expression is compared against all of the tags possessed by the object. For picks, this includes the effects of all dynamically assigned and deleted tags. The Restriction tag expression is only invoked when the user triggers the portal to display the list of options to choose from. Consequently, the tags used for picks are the final set of tags after evaluation completes. This also means that picks can dynamically change their state and change whether they appear as valid choices within a given portal.

An excellent example of where the Restriction tag expression comes in handy is with the d20 System. In d20, spells may be memorized multiple times, so the spells cannot be designated as unique. However, a wizard's spellbook will only possess a single instance of each spell the wizard knows. A Restriction tag expression on the spellbook table ensures that spells are unique within the spellbook.

SECONDARY TAG EXPRESSION

The role of the Secondary tag expression is very different from most other tag expressions. It is always used in conjunction with a portal that adds a pick to a container, such as a dynamic table or chooser. Instead of being processed as part of the portal, the tag expression is attached to the pick. When the pick is selected and added to the container, the Secondary tag expression is also added to the new pick. At that point, the tag expression becomes part of the pick for the rest of the pick's existence.

Once added to the pick, the Secondary tag expression behaves just like a Container tag expression. In fact, the name of the mechanism derives from its use, as it becomes a secondary container tag expression. The Secondary tag expression must be satisfied in order for the pick to remain "live". If the tag expression stops being satisfied, the pick becomes non-live and is treated as if it no longer exists within the container.

The Secondary tag expression is always applied against the container to which the pick was added. The tag expression is compared against all of the tags possessed by the container, which includes the effects of all dynamically assigned and deleted tags. The Secondary tag expression is invoked every evaluation cycle at the timing specified. This means that the pick can transition between live and non-live in dynamic fashion, depending on changes that occur with the container.

It's also valid to omit the timing for a Secondary tag expression and use default timing. The default timing is Setup/5000. However, you can setup whatever you want as the default timing for the Secondary tag expression within the definition file.

As a concrete example, consider the D&D 4th Edition game system. If the character selects the appropriate feats to multi-class, he may then choose powers from the new class. However, after those powers are added, what happens if the user deletes the multi-class feat? The separate table for selecting the additional powers disappears, so there is no way for the user to delete the powers. The easy solution is to assign a Secondary tag expression to the table of multi-class powers that ensures all powers added require that the character be multi-classed. This way, if the multi-class feat is deleted, all of those powers that rely on the feat will fail the Secondary tag expression and be treated as if they were never selected.

EXISTENCE TAG EXPRESSION

The role of the Existence tag expression is very similar to the Secondary tag expression. It is always used in conjunction with a portal that adds a pick to a container, such as a dynamic table or chooser. Instead of being processed as part of the portal, the tag expression is attached to the pick. When the pick is selected and added to the container, the Existence tag expression is also added to the new pick. At that point, the tag expression becomes part of the pick for the rest of the pick's existence.

Once added to the pick, the Existence tag expression governs whether the pick continues to exist within its container (hence the name). The Existence tag expression must be satisfied in order for the pick to continue its existence and avoid being automatically deleted. If the tag expression stops being satisfied, the pick is immediately deleted from the container, just as if the user had deleted it. This is a much more severe action than the Secondary tag expression, which simply causes the pick to go non-live. However, it is sometimes the better choice.

The Existence tag expression is always applied against the container to which the pick was added. The tag expression is compared against all of the tags possessed by the container, which includes the effects of all dynamically assigned and deleted tags. The Existence tag expression is invoked every evaluation cycle at the timing specified. This means that the pick can fail the tag expression at any time, depending on changes that occur with the container. Once that occurs, the pick is automatically deleted in its entirety.

It's also valid to omit the timing for an Existence tag expression and use default timing. The default timing is Setup/5000. However, you can setup whatever you want as the default timing for the Existence tag expression within the definition file.

As an example, consider the wizard class within the d20 System. When a level of wizard is added, the user can add all sorts of spells to the wizard, both within the spell book and as memorized spells. Once those spells are added, they are independent of the wizard class level pick. This means that, if the user deletes the wizard level, all of the spells will still exist on the character. You could use a Secondary tag expression for this, but there would be no way for the user to delete these spells if the wizard class tab disappears. It would also be a real nuisance for the user to delete all of the spells, one at a time. If you use an Existence tag expression on the spells, then they will all be instantly discarded when the user delete the last wizard class level pick.

HOLDLIMIT TAG EXPRESSION

The role of the HoldLimit tag expression is to determine whether a piece of gear can be held by another piece of gear. By default, any piece of gear can be held within gear designated as a "holder". With the HoldLimit tag expression, the set of gear that can be held by a holder is further restricted. Only a piece of gear that also satisfies the HoldLimit tag expression is allowed to be held by the holder.

The primary use of the HoldLimit tag expression is to leverage the containment mechanism to associate weapon options with the various weapons. For example, a laser sight might confer bonuses to the use of a gun, but that laser sight can be moved between different weapons. Through the use of the HoldLimit tag expression, the gun can be restricted to only hold laser sights and other similar weapon options. The user can then move the laser sight gear from one gun to another, with a script being used on the laser sight to automatically apply the appropriate bonus to the gun the laser sight is attached to.

The HoldLimit tag expression is only applied against tags that are assigned directly to a thing at definition, including any component tags. Any tags that are assigned dynamically after the gear is added to the character are ignored. Consequently, the tag expression is evaluated against each piece of gear when the user brings up the menu to move the gear to a new holder. Once a piece of gear is assigned to a holder, no further evaluation is performed, unless the user chooses to move the gear again.

SCRIPTING LANGUAGE OVERVIEW

The scripting engine is a fundamental element of the Kit and you will find yourself writing numerous scripts to successfully manage character data and present it to the user. The Scripting Language section provides in-depth details on how to utilize the scripting language for your data files, and authors should be familiar with all the concepts presented herein before trying to write any scripts. The specifics of individual scripts and details of accessing the information within particular objects will be found in the sections that follow this one.

The scripting language documentation assumes that readers have a minimal familiarity with programming concepts (e.g. the notion of variables). As such, this documentation is not designed for a person with zero understanding of programming. However, anyone that has picked up a programming book and spent a couple of days toying around with a programming language should have plenty of background to work with HL scripts.

IMPORTANT! The Scripting Language documentation assumes you are already familiar with the basics of scripting and the evaluation cycle. If you have not already done so, please review the section Data Manipulation Basics before proceeding with these topics.

IMPORTANT! The scripting language and parsing mechanisms used by the Kit are relatively simple. This means that certain features provided in more complex programming languages are not available, and this is intentional. Complexity is not needed to support the writing of typically short and simple scripts. In addition, a simple language makes it relatively easy for non-programmers to modify existing scripts and/or write their own. All the intricacies of multiple, highly complex game systems have already been implemented using the scripting mechanisms provided within the Kit, so you should have everything you need to fully develop data files for whatever game system you set your sights on.

LANGUAGE SYNTAX

The scripting language used by the Kit is not a clone of another language, although it does share similarities to a few. This section outlines the basic syntactic rules that govern the language.

The scripting language is line-based. Exactly one statement must be on a line, and each line is terminated by a carriage return (or newline). Long statements may not be split across multiple lines.

Any script line on which the very first non-whitespace character is a '~' is treated as a comment and ignored. Use of the '~' character anywhere else on a line (i.e. after the first non-whitespace character) is treated as normal scripting code. Therefore, it is not possible to specify "end-of-line" comments.

All script code is case-sensitive. Uppercase letters are distinct from lowercase characters in all circumstances. Therefore, the variable "foo" is distinct from the variables "Foo" and "FOO". This applies to all facets of the scripting language, which uses lowercase text exclusively for all keywords (e.g. "var", "if", "then", etc.).

DECLARING VARIABLES

Variables can be declared to store data for subsequent use within the script. Variables have the same naming rules as unique ids: maximum of 10 characters, limited to alphanumeric characters and the underscore, and cannot start with a digit.

Variables can be either a number (tracked as a floating point value internally) or a string of text. String variables have a maximum size limitation of 5000 characters, which should be more than adequate for any practical purpose.

Variables are declared with the syntax "var name as type", where "name" is the variable name to declare and "type" is the variable type. Valid values for the type are: "string" for strings and "number" for numeric values.

Examples of legal variable declarations are shown below:

```
var temp as number
var Word as string
var IN_CAPS as number
```

Examples of illegal or unwise variable declarations are shown here:

```
var name_too_long as number (over 10 character limit)
var one.two as string (illegal character used)
var 123456x as number (legal but unwise - starts with a digit)
var NUMBER as number (legal but unwise - don't rely on case to distinguish)
```

Within a script, variables are simply referenced using their name. Once a specific variable is declared to be of a particular type, it cannot be declared as another type. However, it is valid for the variable "foo" to be declared multiple times with the same type, as long as subsequent declarations are of the same type. If the variable already exists with the same type, the new declaration is simply ignored. This means that the following code is perfectly legal (though redundant):

```
var foo as number
var foo as number
```

However, the next block of code is not legal:

```
var foo as number
var foo as string
```

Every time a script is evaluated, it starts with a "clean slate". Therefore, values of variables do not persist from one invocation to the next. Once a variable is declared, it is initialized with a default value. For numeric variables, the default value is zero. For strings, the default value is an empty string ("").

Variables can be declared at any time within a script. The only requirement is that a variable must be declared before it is actually used. It is perfectly valid to declare a variable immediately before it is first used.

BASIC LANGUAGE MECHANISMS

The scripting language supports the following basic language mechanisms:

- **Assignment:** Variables and attributes can be assigned values from other variables, attributes, and literal values. Assignment statements typically take the traditional form "x = y".
- **Numeric Assignment:** Four additional assignment behaviors are supported that are exclusive to numeric values. The first two are the increment ("+=") assignment and decrement ("–=") assignment. Both are essentially a shorthand notation for the statements "x=x+y" and "x=x–y", respectively. Consequently, the statement "x+=y" is equivalent to "x=x+y". The other two are the multiply ("*=") assignment and divide ("/=") assignment, which are a shorthand for the statements "x=x*y" and "x=x/y", respectively. This assignment syntax can be incredibly convenient when identifiers are used with lengthy context transitions and target references.
- **String Assignment:** One additional assignment behavior is supported that is exclusive to string values. It is the concatenate ("&=") assignment, which is essentially a shorthand notation for the statement "x=x&y". Consequently, the statement

"x&=y" is equivalent to "x=x&y". This can be incredibly convenient when identifiers are used with lengthy context transitions and target references.

- **Arithmetic Expressions:** Complex arithmetic expressions are fully supported by the scripting language. All of the standard operators are supported (+–*/), as is the use of parentheses to control the order of evaluation in an expression. An additional operator (%) is supported, which returns the remainder after dividing "x" by "y" (also called the "modulus").
- **String Expressions:** Strings can be concatenated together through string expressions. To concatenate two strings, use the '&' operator (e.g. "str1 & str2"). No other string operators are supported.
- **Implicit Type Casting:** The scripting language will automatically convert a variable or attribute between the two types when necessary. If you assign a numeric variable to a string, the value is converted to a string automatically (e.g. the value 521 is converted to the string "521"). Conversely, assigning a string variable to a number converts the string before the assignment takes place. However, going from a string to a number only converts the leading numeric portion of the string, so the string "1234hello789" is converted to the value 1234. **IMPORTANT!** Type casting only occurs with individual variables/attributes and not with entire expressions. Therefore, it is not valid to assign an arithmetic expression to a string, although it is valid to use a string variable within an arithmetic expression. If you want to assign an arithmetic expression to a string variable, you must first assign the expression result to a numeric variable and then assign that variable to the string. Similarly, it is valid to use a string variable as a parameter to an intrinsic function where a number is required, or vice versa.

The following sample code demonstrates the basic language mechanisms in use:

```
~Declare a variable and put a number into it.
 var myint as number  myint = 7


~Declare another variable and set it equal to an arithmetic expression.
 var temp as number
 temp = (myint + 100) * 3


~Decrement temp by the value of "myint" times three.
 temp -= myint * 3


~Set the value of a string to a numeric value, converting it to a string.
 var mystr as string
 mystr = temp


~Append the contents of two strings
 mystr = "123" & mystr


~Append text to the end of the current string
 mystr &= "123"


~Set a numeric variable to a string, automatically converting the result.
 myint = mystr
```

FLOW CONTROL

The scripting language supports a variety of basic flow control mechanisms, and each is described in the sections below.

LABEL AND GOTO STATEMENTS

The most primitive form of flow control is the use of "label" and "goto" statements. Label statements are defined with the form "name:", where "name" follows the same naming rules as variables. To transfer control to a label statement, a goto statement is used. The syntax for a goto statement is "goto name", where name corresponds to an existing label within the script. When a goto statement is executed, control passes to the label statement, and the next line executed in the script is the one immediately following the label statement. It is perfectly valid to issue a goto statement prior to the definition of the label statement, allowing you to skip over a section of the script if you wish. An example code block demonstrating the use of labels and gotos is given below.

```
var foo as number
foo = 10
 goto mylabel
 foo = 20
 mylabel:
```

```
~The value of foo is still 10, since we skipped over the assignment to 20.
```

IF/THEN BLOCKS

A more traditional form of control flow is with the "if/then" block. Within an if/then block, a test of some sort is performed. Based on the results of that test, the following block of code may or may not be executed. The if/then block consists of two separate statements. The first statement is the test to be performed and uses the syntax "if (expr1 comparison expr2) then". The values of "expr1" and "expr2" can be any valid arithmetic expression. The valid values for "comparison" are the various relationship operators: '<', '<=', '=', '>=', '>', and '<>'. The first five of these comparison operators should be familiar, while the last one implies "not equal".

The "if" statement evaluates the two expressions and then compares them with the relationship indicated. If the comparison is satisfied, then execution continues with the line immediately following the "if" statement.

The second statement identifies the end of the if/then block and has the simple syntax "endif". If the comparison is failed, then execution skips over the if/then block until the "endif" statement is reached, at which point execution resumes. An example if/then block is shown below.

```
var foo as number
 foo = 10
 if (foo + 3 > 20) then
   foo = 1
   endif
 ~The value of foo is still 10, since we skipped over the assignment above.
```

IF/THEN/ELSE BLOCKS

The if/then block can be extended to be an if/then/else block. This form is useful when you need to execute one block of code if the condition is true and another block of code if it fails. The if/then/else block is identical to the if/then block, except that an additional "else" statement is inserted. This new statement demarcates where the block of code to execute on success ends and the block of code to execute on failure begins. If the conditional test is satisfied, then execution continues with the next line, but when the "else" statement is encountered, execution jumps to the line following the "endif" statement. If the conditional test fails, then execution jumps to the line following the "else" statement. An example if/then/else block is shown below.

```
var foo as number
 foo = 10
 if (foo + 3 > 20) then
   foo = 1
 else
   foo = foo + 10
   endif
 ~The value of foo is now 20, since we executed the "else" clause only.
```

IF/THEN/ELSEIF/ELSE BLOCKS

To handle the situation where multiple value ranges need to be handled, the if/then/else block also supports an "elseif" statement. The "elseif" statement allows a separate condition to be tested, with its own block of code to be executed if the condition is satisfied. All condition blocks are tested in sequence, with only the first one that is satisfied having its code executed.

```
var foo as number
 foo = 8
 if (foo <= 5) then
   foo = 1
 elseif (foo <= 10) then
   foo = 2
 elseif (foo <= 15) then
   foo = 3
 else
   foo = 4
   endif
 ~The value of foo is now 2, since we executed the first "elseif" clause only.
```

WHILE/LOOP BLOCKS

Another control flow mechanism provided by the scripting language is the "while/loop" block. The while/loop block executes a block of code repeatedly as long as a particular conditional test remains satisfied. The while/loop block begins with a statement of the form "while (expr1 comparison expr2)", where "expr1", "expr2", and "comparison" all behave identically to a standard "if" statement. The end of a while/loop block is identified by a "loop" statement, which consists solely of the keyword "loop" on a line. When the "loop" statement is

encountered, the conditional test is evaluated again. As long as the condition remains satisfied, execution continues with the line following the "while" statement. Once the condition fails, execution jumps to the line following the "loop" statement. The loop will continue executing the code until the conditional test finally fails. An example while/loop block is shown below.

```
var foo as number
 foo = 1
 while (foo <= 10)
   foo = foo + 1
   loop
 ~The value of foo is now 11.
```

FOR/NEXT BLOCKS

The final control flow mechanism available is the "for/next" block. Like the while/loop block, the for/next block executes a block of code repeatedly, keying on the value of a variable (the "loop index"). The for/next block is identified by "for varname = expr1 to expr2", where "varname" is the name of the numeric variable to be used as the loop index and "expr1" and "expr2" are complex expressions. The variable is initialized with a given value and incremented by 1 every iteration of the loop. After the second value has been reached and processed, the iteration stops (so looping from 1 to 5 will run through the loop 5 times, the index taking on values of 1, 2, 3, 4 and 5 in succession before stopping). The loop index is automatically incremented by the for/next construct itself, and does not have to be maintained by the script writer.

```
var x as number
 var text as string
 text = "A list of some numbers: "
 for x = 0 to (100 / 20 * 2)
   text = text & x
   next
 ~The string 'text' will now contain an ascending list of the numbers from 0 up to 10.
```

NESTING

All conditional statements (if/then, if/then/else, while/loop and for/next) can be safely nested within each other. For example, if two separate conditions must be satisfied in order to perform some action, you might have a script that looks similar to the example below.

When nesting conditional statements, each "endif" statement is associated with the closest "if/then" statement. Appropriate use of indentation can greatly enhance the readability of scripts when nested statements are employed.

```
if (x > y) then
   if (a > b) then
     z = 1
   else
     z = 0
     endif
 else
   if (a > b) then
     z = 0
   else
     z = 1
     endif
   endif
```

IMPORTANT! The scripting mechanism is designed to handle scripts of only moderate size and complexity. As such, scripts possess a maximum nesting depth of 20 levels, which ought to be significantly more than any script should ever need.

If a script is too large or complex, an error will be reported when compiling the script. In general, a given script can utilize dozens of flow control constructs without becoming too complex, so the complexity limit should not be reached under normal conditions.

DONE STATEMENT

If you reach a point in the execution of a script where all necessary processing is completed, you can utilize the "done" statement to immediately exit the current script. Judicious use of "done" can simplify scripts and make them easier to maintain by eliminating a great deal of nesting and matching of if/then/else statements.

Two examples are shown below, where the second script accomplishes the same as the first. The key difference is that the second script utilizes the "done" statement. Use of the "done" statement is a matter of personal style, but it can be very handy for scripts with both simple and complex conditions.

```
if (foo <= 3) then
   ~do something  else
```

```
  ~lots of code
  if (for <= 6) then
    ~lots of code
  else
    ~lots of code
    endif
  ~lots of code
  endif
```

```
if (foo <= 3) then
  ~do something
  done
  endif
~lots of code
if (for <= 6) then
  ~lots of code
else
  ~lots of code
  endif
~lots of code
```

DONEIF STATEMENT

There will be numerous situations when writing scripts where you'll want to test for a condition and exit out of the script. This amounts to writing an "if" statement and using "done" if the test is satisfied. That's three lines of code that you'll be using over and over again.

To simplify this, the scripting language provides a "doneif" statement. This statement takes a comparison expression and will exit the script if the test is satisfied, thereby reducing three lines of code to one. For example, the following two blocks of code are functionally equivalent.

```
if (foo <= 3) then
  done
  endif
```

```
doneif (foo <= 3)
```

OTHER LANGUAGE STATEMENTS

The scripting language supports additional statements for special purposes that can be extremely useful.

PERFORM STATEMENT

A large number of target references that apply changes to objects within scripts return a value that you can ignore. For example, the "assign" target reference that assigns a tag to an object always returns the value zero. Putting it to use will normally look like the following:

```
var result as number
result = assign[group.tag]
```

Declaring a variable in which to place a return value that you don't care about is an unnecessary nuisance. So the scripting language provides the "perform" statement. The perform statement tells the compiler to invoke the operation that follows the statement and simply throw away the value returned. The above example changes to the code below when the "perform" statement is employed.

```
perform assign[group.tag]
```

The net result is simpler, shorter, and clearer scripting code.

DEBUG STATEMENTS

If you need to write more than a few, simple scripts, the odds are that something won't work at some point along the way. So Hero Lab includes some very helpful debugging aids. One of these aids is the "debug" statement. The debug statement allows you to have Hero Lab

output any information you can access via scripts to the screen during the evaluation process. When a debug statement is executed, the string you specify is evaluated and then output to the special debug information window within HL, where you can view the results and determine what changes need to be made to your scripts. An example code block using the debug statement is provided below.

```
var foo as number
foo = 10
var bar as string
bar = "hello"
debug "foo = " & foo & " and bar = [" & bar & "]"
```

The final output from the above script code will be as shown below within the debug info window.

```
foo = 10 and bar = [hello]
```

Using the debug statement requires that you utilize the info windows within HL. To view the debug output, go to the "Develop" menu within HL, select the "Floating Info Windows" option, and then select the "Show Debug Output" sub-option. This will display a window which contains the debug information that has been output from your scripts.

As your scripts continue executing and the debug window fills up, the oldest information will be lost off the top to keep memory consumption to a minimum.

FOREACH STATEMENT

There will be times when you need to iterate through a collection of objects, separately processing each individual object within the collection. For example, you might want to iterate through all the weapons possessed by the character to determine how many "hands" worth of weapons are equipped by the character.

In situations like this, the "foreach" statement provides the perfect solution. The foreach statement requires that you specify something to iterate over. The loop is terminated by the "nexteach" statement, resulting in the code within the foreach/nexteach block being invoked for every instance that satisfies the foreach criteria. The net result is that a typical use of a foreach/nexteach block looks like the code below.

```
foreach item in context where tagexpr sortas sortset
    ~code goes here
    nexteach
```

In the above code, the "foreach" statement stipulates what type of object is to be iterated ("item"), followed by the "in" designation and an appropriate context to search within ("context"). An optional tag expression can be specified that restricts the set of items iterated through by specifying the "where" clause and a string that contains the tag expression ("tagexpr"). The list of items can also be sorted via use of the "sortas" clause that specifies the unique id of the sort set to be used ("sortset").

There are multiple versions of the "foreach" statement that are supported. All versions work similarly, although each version iterates through a different assortment of items and context. The various versions are detailed in the sections below.

"FOREACH PICK IN CONTAINER"

This form of "foreach" iterates through the picks that have been added to a container, processing only those that satisfy the specified set of criteria. With this version, the "context" must specify the container, which can be any valid context transition that identifies a container (e.g. "hero" or "child[pickid].gizmo"). The code within the "foreach" block is invoked for every pick in the container that satisfies the optional tag expression. Within the "foreach" block, the current pick being iterated can be accessed via the "eachpick." script context transition. The net result is that a typical use of this version of "foreach" looks like the code below.

```
foreach pick in hero where "group.tag"
    debug "id: " & eachpick.idstring
    nexteach
```

The above code will iterate through all picks within the hero and identify all that possess the "group.tag" tag. The "debug" statement (see below) will then output the unique id of each of the picks that are processed by the "foreach" statement, since the "eachpick." script context specifies access to the current pick within the body of the loop.

"FOREACH THING IN COMPONENT"

This form of "foreach" iterates through all things that derive from a specific component and satisfy any additional criteria. The "context" must specify the unique id of a component. The code within the "foreach" block is invoked for every thing derived from that component that also satisfies the optional tag expression. Within the "foreach" block, the current thing being iterated can be accessed via the "eachthing." script context transition. The net result is that a typical use of this version of "foreach" looks like the code below.

```
foreach thing in compid where "group.tag"
    debug "id: " & eachthing.idstring
    nexteach
```

The above code iterates through all things derived from the component with the unique id "compid" and that possess the "group.tag" tag. The "debug" statement outputs the unique id of each thing that is processed by the "foreach" statement.

"FOREACH BOOTSTRAP IN THING"

This form of "foreach" iterates through all things that are directly bootstrapped by a specific thing and satisfy any additional criteria. The "context" must specify the thing, which can be any valid context transition that identifies a thing or a pick. The code within the "foreach" block is invoked for every thing that is directly bootstrapped by the identified thing that also satisfies the optional tag expression. This includes both bootstraps defined on the thing and on any components the thing derives from. Within the "foreach" block, the current bootstrap thing being iterated can be accessed via the "eachthing." script context transition. The net result is that typical uses of this version of "foreach" looks like the code below.

```
foreach bootstrap in this where "group.tag"
    debug "id: " & eachthing.idstring
    nexteach
```

The above code iterates through all things that are bootstrapped by the current thing/pick context ("this") and that possess the "group.tag" tag. The "debug" statement outputs the unique id of each thing that is processed by the "foreach" statement.

NOTE! This mechanism is primarily intended for use within scripts that synthesize description text for a thing.

"FOREACH BOOTSTRAP IN ENTITY"

This form of "foreach" iterates through all things that are directly bootstrapped by an entity and satisfy any additional criteria. The "context" must always be the literal string "entity", and it limits this statement to use within things that possess a child entity. The code within the "foreach" block is invoked for every thing that is directly bootstrapped by the child entity that also satisfies the optional tag expression. This includes both bootstraps defined on the entity definition and via the "child" element that attaches the entity. Within the "foreach" block, the current bootstrap thing being iterated can be accessed via the "eachthing." script context transition. The net result is that a typical use of this version of "foreach" looks like the code below.

```
foreach bootstrap in entity where "group.tag"
    debug "id: " & eachthing.idstring
    nexteach
```

The above code iterates through all things that are bootstrapped by the child entity of the current thing/pick context and that possess the "group.tag" tag. The "debug" statement outputs the unique id of each thing that is processed by the "foreach" statement.

NOTE! This mechanism is primarily intended for use within scripts that synthesize description text for a thing.

"FOREACH ACTOR IN PORTFOLIO"

This form of "foreach" iterates through the actors that exist within the portfolio, processing only those that satisfy the specified set of criteria. With this version, the "context" must always be the literal string "portfolio". The code within the "foreach" block is invoked for every actor in the portfolio that satisfies the optional tag expression. Within the "foreach" block, the current pick being iterated is always the "actor" pick for the current actor and can be accessed via the "eachpick." script context transition. The net result is that a typical use of this version of "foreach" looks like the code below.

```
foreach actor in portfolio where "group.tag"
    debug "name: " & eachpick.hero.name
    nexteach
```

The above code will iterate through all actors within the portfolio and identify all that possess the "group.tag" tag. The "debug" statement (see below) will then output the name of each actor that is processed by the "foreach" statement, since the "eachpick." script context specifies access to the current pick within the body of the loop and the parent actor is then used.

NOTE! This mechanism is primarily intended for use within the Dashboard and Tactical Console, where all the actors must be managed appropriately.

NOTIFY STATEMENT

In order to facilitate debugging and to provide a convenient means for data files to report special events to the user, the "notify" statement is provided. The notify statement works similarly to the "debug" statement, except that the resulting string is reported directly to the user

via a message alert. For all practical purposes, the syntax for the "notify" statement is identical to the "debug" statement, except that the line starts with "notify", as shown below.

```
notify "This is the message displayed"
```

The notification is queued and then reported to the user at the first reasonable opportunity. This means that the message is not necessarily reported to the user immediately when it is triggered. Similarly, if there are multiple notification messages, they will be accumulated by HL and then reported collectively at the next opportunity. The messages will be displayed within a single report to the user, with each message on a separate line.

APPEND STATEMENT

The "append" statement is used exclusively with the Synthesize script. When generating dossier output, the volume of text can be quite long, in which case the traditional "@text" special symbol becomes impractical and inefficient. So the append statement is provided as a means to output a chunk of text to HL and then allow the author to stop worrying about it. Once text is output via the append statement, it will be part of the synthesized output, and it will appear in the sequence it is output.

The syntax for the append statement is simple, as it uses a single string. When an append statement is executed, the string you specify is evaluated and then output as part of the dossier. This means that three append statements in a row will procedure output that is the concatenated result of all three strings, as shown below.

```
append "line #1" & @newline
append @boldon & "line #2" & @boldoff & @newline
append "line #3" & @newline
```

The final output that results from the above three statements would look like the following when synthesized for HTML:

```
line #1<br>
<b>line #2</b><br>
line #3<br>
```

TRUSTME STATEMENT

As a general rule, scripts should not be directly modifying the contents of "user" fields. Those fields are intended for access only by the user, so attempts to modify them via scripts are inherently considered an error by the compiler. However, as with any rule, there are always the exceptions.

When a script needs to modify the value of a user field, the "trustme" statement can be used. This tells the compiler that you know what you're doing and can be trusted to not do things inappropriately. Once specified within a script, the script becomes trusted and the compiler stops checking for code that modifies user fields.

The syntax of the trustme statement is trivial. Simply place the statement on its own line within the script. There are no parameters, so it looks like below.

```
trustme
```

IMPORTANT! Whenever you think that you need to utilize the trustme statement, look closely at your script and what you're trying to do. It is rare that you should actually need to use trustme, and your data files probably can be better structured to eliminate the need for directly modifying a user field.

IMPORTANT! The following script types are automatically designated as "trusted". This is because their general nature is such that you'll often be needing to modify user fields.

- Trigger Script
- Creation Script
- TransactSetup Script
- TransactBuy Script
- TransactSell Script
- Merge Script
- Split Script
- NewCombat Script
- NewTurn Script
- Integrate Script
- Initiative Script

- LoadFixup Script

LANGUAGE INTRINSICS

The scripting language has an assortment of intrinsic (i.e. built-in) functions that can be used for various purposes. Some intrinsic functions operate on strings, while others operate on numbers. Some intrinsic functions return strings, while others return numbers.

The purpose of intrinsic functions is to provide authors with re-usable mechanisms that can be used to perform script operations that arise on a recurring basis. For example, the Kit includes intrinsic functions for searching string, carving up strings, comparing strings, and replacing text within strings. Intrinsics are also included to determine the minimum or maximum of two values, round a number off at a certain level of precision, and generate a random number.

INTRINSIC FUNCTIONS

The complete list of intrinsic functions provided by the language is presented below.

length	number length(string str)	Returns the number of characters in the string str.
left	string left(string str, number num)	Returns a new string containing the leftmost num characters of the string str.
right	string right(string str, number num)	Returns a new string containing the rightmost num characters of the string str.
mid	string mid(string str, number start, number num)	Returns a new string containing a substring of string str. The new string begins with the character at position mid start and is num characters in length. Character positions are 0-based, so the first character in a string is at position zero (0).
pos	number pos(string str, string search)	Return the character position where the string search first appears within str. Character positions are 0-based, so the first character in a string is at position zero (0). If search does not exist within str, a value of –1 is returned.
uppercase	string uppercase(string str)	Returns a new string that is an upper case version of string str.
lowercase	string lowercase(string str)	Returns a new string that is an lower case version of string str.
lastpos	number lastpos(string str, string search)	Return the character position where the string search last appears within str. Character positions are 0-based, so the first character in a string is at position zero (0). If search does not exist within str, a value of –1 is returned.
asc	number asc(string str)	Returns the ASCII value of the first character of string str.
chr	string chr(number val)	Returns a new string that consists of a single character, which has an ASCII value of val. For example, 'chr(10)' is the newline character.
compare	number compare(string str1, string str2)	Compare the two strings str1 and str2. If the strings are identical, the value 0 is returned. If str1 would appear before str2 in an alphabetical sort (based on the ASCII code of each character), a value less than 0 is returned. If str1 would appear after str2 in an alphabetical sort, a value greater than 0 is returned. Note that all uppercase characters are sorted before lowercase characters.
replace	string replace(string str, string match, string replace, number maxcount)	Searches through the string str and replaces all instances of the string match with the string replace. A maximum number of replacements is given by maxcount, with a value of zero indicating that all matches should be replaced. The converted string is returned, with the original string left untouched.
int	number int(number val)	Returns a value that represents the integer portion only of val . For example, the integer portion of the value – 123.45 would be – 123.
random	number random(number range)	Returns a random integer value between zero and range–1.
minimum	number minimum(number val1, number val2)	Returns the lower of the two values given.

maximum	number maximum(number val1, number val2)	Returns the higher of the two values given.
power	number power(number val1, number val2)	Returns val1 raised to the power of val2 (e.g. x^y). For example, "power(4,2)" would yield 4 squared, which is 16.
nthroot	number nthroot(number val, number nth)	Returns the nth root of a number, where val is the value to get the root of and nth indicates the root to obtain. For example, "nthroot(16,2)" would yield the square root of 16, which is 4.
round	number round(number val, number dec, number dir)	Returns the rounded value of val, where dec indicates the number of decimal places at which to perform the rounding and dir indicates how to perform the rounding. If dir is zero, normal rounding is performed (e.g. 0.5- 0.99 round up to 1.0 and 0.01-0.49 round down to 0.0). If dir is positive, the value is always rounded up. If dir is negative, the value is always rounded down. For example, "round(4.36,1,0)" would yield 4.4 and "round(4.36,1,-1)" would yield 4.3.
signed	string signed(number val)	Returns a string that represents the properly signed version of 'val', including a prefix of either '+' or '-'. For example, the signed version of the value 1.42 would be "+1.42", while the signed version of the value -6.23 would be "-6.23".
decimals	string decimals(number val, number dec)	Returns a string that contains the rounded value of val, where dec indicates the number of decimal places at which to perform the rounding (normal rounding is always performed). In addition, if the value requires fewer decimal places than specified, zeroes are appended to bring the value to the full number of decimals. This intrinsic is ideal for formatting currency with a fixed number of decimals, so that "decimals(1.5,2) produces "1.50" instead of the normal "1.5".
bitwise_and	number bitwise_and(number val1, number val2)	Returns the bit-wise "and" of val1 and val2, treating both parameters as integer values for the operation. The bit-wise "and" performs a comparison of the two values, one bit at a time, with the resulting value having a bit value of 1 only if both val1 and val2 have that bit set to 1. For example, "bitwise_and(14,7)" would yield 6, since the two parameters have binary representations of "1110" and "0111", which results in a value with a binary representation of "0110" (or 6).
bitwise_or	number bitwise_or(number val1, number val2)	Returns the bit-wise "or" of val1 and val2, treating both parameters as integer values for the operation. The bit-wise "or" performs a comparison of the two values, one bit at a time, with the resulting value having a bit value of 1 if either val1 or val2 has that bit set to 1, or if both have the bit set to 1. For example, "bitwise_or(10,3)" would yield 12, since the two parameters have binary representations of "1010" and "0011", which results in a value with a binary representation of "1011" (or 12).
bitwise_xor	number bitwise_xor(number val1, number val2)	Returns the bit-wise "xor" of val1 and val2, treating both parameters as integer values for the operation. The bit-wise "xor" performs a comparison of the two values, one bit at a time, with the resulting value having a bit value of 1 only if one of val1 and val2 have that bit set to 1 - not both. For example, "bitwise_xor(14,7)" would yield 9, since the two parameters have binary representations of "1110" and "0111", which results in a value with a binary representation of "1001" (or 9).
bitwise_not	number bitwise_not(number val)	Returns the bit-wise "not" of val, treating the parameter as an integer value for the operation. The bit-wise "not" processes the value, one bit at a time, with the resulting value having each bit possess the opposite value that it started with. This means that any bit with a value of 1 becomes 0, and vice versa. For example, “bitwise_not(9)" would yield 6, since the parameter has a binary

		representation of "1001", which results in a value with a binary representation of "0110" (or 6).
empty	number empty(string str)	Returns a non-zero value if the string str has a length of zero characters, else a value of zero is returned. The "empty" intrinsic is significantly faster than using "length" and comparing it to zero.
plaintext	string plaintext(string str)	Returns the string str with all encoded text stripped out. This provides a quick an easy mechanism for converting a string with encoded text (e.g. color usage) to its equivalent without any special formatting.
today	number today()	Returns the current date in the proper format utilized by fields that store dates and times which users can edit via "editdate" portals. This provides a convenient mechanism for initializing journal entries and any other situations with the current date.

EXAMPLES

Simple examples of how to use each of the various intrinsic functions are provided below.

```
var str as string
var len as number
var piece as string
var temp as string
var val as number
var total as number
var NewLine as string
var tag_count as number
var link_count as number
str = "hello there world there"


~Get the length of the string, which is 23
len = length(str)


~Extract the first 5 character of the string, which is "hello"
piece = left(str,5)


~Extract the last 5 characters of the string, which is "there"
piece = right(str,5)


~Extract 5 characters from the string, starting at position 6, which is "there"
piece = mid(str,6,5)


~Locate the first occurrance of the string "there" within the string, which
~starts at position 6
val = pos(str,"there")


~Locate the last occurrance of the string "there" within the string, which
~starts at position 18
val = lastpos(str,"there")


~Convert a string to all uppercase, then to all lowercase
temp = uppercase(str)
temp = lowercase(str)


~Get the ASCII value of the first character of the string
val = asc(str)


~Create a string of one character, where that character is an 'A'
piece = chr(65)
```

```
~Create a string of one character, where that character is a newline
NewLine = chr(10)


~Compare the two strings, with the first string being less than the second
val = compare(str,"second")


~Replace all instances of "there" with "change"
temp = replace(str,"there","change",0)


~Extract the integer portion only of –123.45, which is –123
val = int(-123.45)

~Generate a random number between 0 and 9
val = random(10)


~Determine the lower and higher of two values
val = minimum(123.45,val)
val = maximum(123.45,val)


~Calculate 4 squared, which yields 16
val = power(4,2)


~Calculate the square root of 16, which yields 4
val = nthroot(16,2)


~Round the value 4.36 normally to one decimal place, which yields 4.4
val = round(4.36,1,0)


~Get the signed version of a value
temp = signed(-123.45)


~Convert a value to formatted currency
temp = "$" & decimals(1.5,2)


~Perform bitwise operations on two values
total = bitwise_and(14,7)
total = bitwise_or(10,3)
total = bitwise_xor(14,7)
total = bitwise_not(9)


~Determine if a string is empty
value = empty(str)
```

SPECIAL SYMBOLS

Many scripts have special pieces of information that are either passed into the script from HL or that the script must specify for subsequent use by HL. For example, a script-based label portal has two such values. On entry, the script is informed whether it is being applied to either a thing or a pick, since that distinction is important for some scripts. On exit, the script must tell HL what the final text is that should be displayed within the label.

Whenever a value needs to be passed into or out of a script, a "special symbol" is used. Special symbols behave exactly like variables, so they can be used just like a variable within the script. The name of each special symbol is always prefixed with the '@' character, which is followed by the name of the special symbol. For example, to access the special symbol "value", the syntax would be "@value".

The set of special symbols varies for each script, as will their use, with some scripts having zero special symbols and a few having many. The exact symbols for a given script are specified with the details for each script within in Kit Reference documentation.

SCRIPT MACROS

The scripting language syntax is designed to support accessing a complex assortment of game elements. This can result in some rather lengthy terms to specify exactly the information you want to access. Many of these terms will be used extensively within the data files for a given game system. To simplify scripts, the Kit allows you to define an assortment of script macros that serve as a shorthand notation for quickly accessing common elements.

For example, in the data files for the d20 System, there are numerous effects that apply bonuses to skills (e.g. feats). The syntax for accessing the bonus value for a given feat looks something similar to the code shown below (for more details, please refer to the Definition File Reference).

```
hero.childfound[kTumble].field[Bonus].value
```

If you had to type this in everywhere, there would be lots of opportunity for errors and it would be tedious to type each time. So the d20 System data files define a "skillbonus" macro that can be used in place of the above syntax. Using the "skillbonus" macro, the resulting syntax is simplified to the following:

```
#skillbonus[kTumble]
```

By using a script macro, the code becomes simpler to write, plus it becomes clearer regarding what it's doing. The above macro is clearly accessing the skill bonus value for the skill with unique id "kTumble". This is much clearer than the lengthy text to which the macro corresponds.

The syntax for using a script macro is that it always starts with a '#' character, followed by the name of the macro. Parameters for the macro are placed within square brackets and separated by commas (if there are more than one). When HL compiles the data files, it knows to replace the macro with the corresponding long syntax, as specified for the game system, allowing the compiler to generate the proper behavior.

The data file structure will vary greatly from one game system to the next, as will the commonly accessed information for each game system. As such, the specific set of macros defined for each game system will vary based on what's most appropriate that game system. To simplify getting started with your own data files, the starter data files provided with the Kit include a number of useful script macros that you can both put to use and refer to as examples when adding your own new macros.

RE-USABLE PROCEDURES

Procedures provide the means to define script logic in a single location and re-use that logic within multiple scripts by calling the procedure from each of those scripts. Procedures are an extremely convenient tool and are used regularly for a variety of purposes.

There are a number of important considerations that come in to play when using procedures. Some are fundamental to the nature of procedures, while others are selectable, allowing you to choose between various capabilities to better tailor the procedure to your needs.

The topics below summarize each of these considerations.

INHERIT CALLER'S CONTEXT

When invoked, procedures inherit the initial script context of their caller. For example, if a Description script calls a procedure, the pick or thing that is the initial context of the script is also the initial context within the procedure. There is one exception to this, which is discussed in the topic on "Any" Procedures (below).

NO PARAMETERS

Unlike most programming languages, procedures have no parameters. You cannot pass parameters to procedures in the way that traditional programming languages do. However, you can pass information between caller and procedure via the use of variables (see below).

INHERIT CALLER'S LOCAL VARIABLES

Procedures inherit all local variables of the calling script. However, you must still declare each local variable within the procedure so that the compiler knows that you intend to use them. When that is done, the value of each variable will be inherited when the procedure is called and can be accessed normally via the variable within the procedure. Using local variables, you can pass information between the calling script and the procedure. Changes made to variables within the procedure will remain in effect upon return to the calling script, so you can use this technique to pass information both from the script into the procedure and back out from the procedure to the calling script upon return.

SCRIPT TYPE VS. CONTEXT

All procedures must be assigned either a specific script type or a more general script context. This assignment dictates important behaviors for the procedure that impact which scripts can legally call the procedure. Scripts can only call procedures that are assigned a compatible classification. Either the script type must match exactly or the script context must match generally. For example, you cannot call a procedure with "container" context from within an Eval Script, since that script has a "pick" context.

There is a distinct advantage to designating a specific script type for a procedure. If a script calls a procedure with the same script type, then the procedure has full access to all of the special symbols defined for the script. The special symbols behave just like inherited variables and are shared back and forth. If a procedure does not share the identical script type, special symbols are not accessible.

There is a different advantage to using a more general script context for a procedure. By using a general script context, a procedure becomes much more versatile. The procedure can be used from multiple different scripts of different types. For many scripts, using a matching context is preferable, since the procedure can then be re-used by more scripts.

The complete list of script types and contexts available for procedures is specified within the Kit Reference documentation.

"ANY" PROCEDURES

There is a special type of procedure that can be called from absolutely any script. In order to make this possible, though, the procedure must make an additional concession to ensure compatability. The procedure possesses no initial script context. This is because the same procedure could be called from different scripts, where each calling script has a different initial context.

The implications of having no context are significant. Without an initial context, the procedure cannot perform transitions to anything based on an initial context. This means that the only things that can be done are to either operate upon variables (which are shared with the caller) or specify a "safe" context and transition from there. For example, the procedure is always free to start with the "hero" context and locate the specific information it needs, since there is always a hero that can be identified and used.

CALLING OTHER PROCEDURES

Procedures **are** allowed to call other procedures. All of the rules above apply to procedures that are called from other procedures. The same variable space is shared, all of the special symbols are shared, and all of the rules regarding script type and context apply equally to any procedure, even one that is not called directly from a script.

DEBUGGING MECHANISMS

The development of a fully functional set of data files for a game system involves a great many steps and there are bound to be a few problems along the way. To handle this, the Kit provides a variety of mechanisms to help you quickly identify problems and diagnose how to fix them. In order to facilitate the resolution of problems, HL includes a variety of built-in mechanisms that can be leveraged when issues arise. Since errors in software are traditionally called "bugs", we refer to these mechanisms as "debugging" aids and the overall process is commonly referred to as "debugging" your data files.

The topics below outline a number of incredibly useful tools and techniques for quickly figuring out what is going wrong within your data files. Familiarize yourself with these debugging mechanisms now, since you will find yourself putting them to use at various points in the development process.

USING INFO WINDOWS

Whatever the game system, HL will be tracking a great deal of information. Portfolios contain multiple heroes and heroes can have child gizmos. Heroes and gizmos contain a large number of picks, and those picks will contain a healthy number of fields. Everything has a wide assortment tags. And don't forget the task list that tracks all of the different operations that are performed on all of these objects. Even if the game system is relatively simple, there will be lots of information, and the volume increases dramatically as the game system complexity goes up.

At any point in time during development, you'll be working on some subset of this information. After you make some changes to the data files, you'll be expecting everything to work. However, if it doesn't work, you'll want to understand why. This is usually achieved by looking at the appropriate subset of information within HL that you are manipulating to see what is happening, which can typically narrow down where the error is lurking.

To help out in these situations, HL provides a bunch of different ways to view the information that is being tracked internally. This is accomplished through "info windows". Each info window is a floating window that is separate from the main HL window and contains specific information about some facet of the current portfolio. Every time that a change is made to portfolio, the contents of each info window are updated, so you can see the direct effects of changes in real-time.

Info windows can be created at any time by going to the "Debug" menu, selecting the "Floating Info Windows" menu, and then choosing the appropriate option from the sub-menu that is presented. Once created, the info window persists until you close it. You can move the info window around and resize it as you deem appropriate.

Each type of info window contains different information that may prove useful in different situations. The table below offers a quick summary of the information provided by each info window.

- **Hero Tags:** Displays a list of all the tags assigned to the active hero. This list includes tags from three different sources. First, any tags dictated by the user's selection of sources and/or rule sets are included. Second, tags that are dynamically assigned to the actor by scripts are included. Lastly, tags that are automatically assigned to minions are included (only if the hero is a minion).
- **Hero Fields:** Displays a list of all the fields assigned to the active hero. Heroes don't technically possess fields, but every hero automatically contains a single "actor" pick that does possess fields. Since the "actor" pick is considered synonymous with the actor in a variety of situations (e.g. the Tactical Console and Dashboard), the fields within the "actor" pick are considered to be "actor fields" (or hero fields).
- **Task List (Active Hero):** The complete set of all tasks for the active hero are listed. The task list includes all actions that are scheduled to occur during the evaluation cycle, such as Eval Scripts, Eval Rules, Bounding scripts, Condition tests, and a host of other operations. Anything that is scheduled to be performed at a specific phase and priority has a corresponding task that will appear in this list.
- **Task List (Full):** This option shows the complete task list for all heroes within the context of the current lead. When minions are used and those minions have timing inter-dependencies with their master, the interplay of all the tasks for all the actors can be critical. In those situations, the full list can be used to help diagnose timing errors between different actors.
- **Selection List:** Displays a list of all of the picks that have been added to the active hero. There are a variety of mechanisms within the Kit that control whether picks are added to an actor, such as bootstraps, condition tests, and live tests. This info window can help you to diagnose whether things are behaving as you intend.
- **Selection Tags:** When you select this option, a list of all of the picks for the active hero is presented. You can select as many picks as you want, after which separate info windows will be created for each pick you specified. The info windows will contain a complete list of all the tags that are assigned to the pick at the conclusion of evaluation. If a tag is deleted at some point during evaluation, it will not appear in this list.
- **Selection Fields:** When you select this option, a list of all of the picks for the active hero is presented. You can select as many picks as you want, after which separate info windows will be created for each pick you specified. The info windows will contain a complete list of all the fields and their values for the pick at the conclusion of evaluation.
- **Debug Output:** This option opens up a window into which debugging output is written during evaluation. This topic is discussed in detail in the following topics.

DEBUG OUTPUT VIA SCRIPTS

In most traditional programming languages, there is the age-old technique of inserting a "print" statement at various points in the code to display the state information. This technique is used to verify that everything is behaving correctly and to diagnose errors that surface. The Kit provides two related mechanisms that offer a similar solution.

THE "NOTIFY" STATEMENT

The first mechanism is the "notify" statement within the scripting language. This statement allows you to synthesize a string and then display it to the user. The string can be anything, such as the contents of fields, the results of an arithmetic expression, or even just a simple error message. So you can display whatever information you need to convey.

The notify mechanism displays an alert message to the user, requiring the user to dismiss the message before continuing. It works very similarly to the way run-time errors are handled. Consequently, the notify mechanism is ideal for reporting unexpected and/or error conditions that are encountered during evaluation. It is also useful for inserting a quick verification of a few values during testing of new functionality.

THE "DEBUG" STATEMENT

The second mechanism is the "debug" statement within the scripting language. Like the notify mechanism, this statement allows you to synthesize a string for output, but the resulting text is not displayed directly to the user. Instead, the text is sent to the Debug Output window, which can be shown via the "Floating Info Windows" sub-menu of the "Debug" menu.

When the Debug Output window is visible, the use of a sequence of "debug" statements will send a progressive series of messages to the window. You can use this technique to monitor the flow of execution through all of your scripts and watch both the values of fields and the presence of tags at various points during the evaluation cycle. Every time the evaluation cycle is triggered, all of the debug messages are output again, so you can easily watch the effects of changing selections and values within the character through each new evaluation cycle.

This particular technique is probably the single most valuable method for quickly diagnosing where things are going wrong within your scripts. Once you know where things are going wrong, it's usually pretty easy to figure out why the error is occurring and put in the proper correction.

SCRIPT TIMING ISSUES

We'll start this section off by answering the obvious question. What's a timing issue?

The evaluation cycle controls the order in which everything is processed for each character, and it's up to the author to make sure that every task is assigned an appropriate phase and priority. A timing issue is the result of scheduling a task in the wrong sequence, which causes the evaluation cycle to yield the wrong final results.

It doesn't matter how experienced you are as an author, script timing issues will crop up if the game system has even a modicum of complexity. The trick is in being able to recognize when you are dealing with a timing issue. In general, timing issues are identified via a process of elimination, wherein the possibility of all other types of errors is systematically ruled out.

This section provides assorted tips on how to recognize and resolve timing issues.

USING THE TASK LIST

The most fundamental tool for solving timing issues is the task list, which is accessible via the "Debug" menu and the "Floating Info Windows" sub-menu. The task list, as its name suggests, presents a single list of all of the different tasks that are scheduled during the evaluation cycle. This list is often rather lengthy, since every pick and every field has its own distinct set of tasks.

The list is sorted in the order that the tasks will be evaluation by the HL engine. For each task, you'll see its phase, priority, and a brief description. This description should help in identifying exactly what the task is doing and what it is operating upon.

If you think you have a timing issue, the thing to do is check the task list. Make sure that each of the tasks you're working with is occurring in the proper sequence. If you're integrating new functionality that relies upon other tasks to setup state appropriately, be sure to check that your new tasks are occurring after the existing tasks have been invoked. Similarly, if your new tasks apply modifications that other tasks will rely upon, make sure to verify that sequencing as well.

Remember that there are two separate task lists. In general, you will want to utilize the task list for the active hero. This task list is perfect for verifying timing relationships between tasks within a single actor, which is normally what you'll be trying to do. The full list contains all the tasks for all actors, interspersed amongst each other. The only time you'll want to use this task list is when you have masters and minions that have inter-dependencies between each other, such as familiars in the d20 System, which have traits derived from their master and also confer bonuses back onto their master.

NOTE! The task list is the **only** way to diagnose timing issues that involve tag expressions, such as live states and condition tests. Using the task list is critical to verifying that important tag expressions are being evaluated appropriately relative to scripts and/or other tag expressions.

DEBUG OUTPUT TRACING

If a review of the task list doesn't reveal the source of the problem, another useful trick is to insert assorted trace information into your scripts. For quick checks, the "notify" statement is convenient. However, in most cases, you're going to find the "debug" statement is preferable.

By sprinkling "debug" statements in various places within scripts, you can achieve two goals. First, you can report important state information during the course of the evaluation cycle. For example, you can output field values and tag lists at various points to ascertain that everything matches the state you expect at each of these junctures.

Second, you can report the overall execution flow of scripts, allowing you to verify that ActionA is actually being triggered and that it's occurring before ActionB. For example, when a field value is set or a tag assigned, you can output a message that relays that fact, with the option of also including details about the new state.

The debug output window displays message in the exact sequence that they are called from within scripts. Consequently, what you see within the debug output window is an accurate reflection of what is actually going on within your data files.

Based on the debug output, you should be able to spot the problem. Sometimes, you'll start with only a handful of debug statements and then add more based on the output you see. This technique will systematically narrow down where the timing problem is occurring until you can finally pinpoint it and determine a proper fix.

SCRIPT TIMING OUTPUT

There are times when you have numerous different tasks all working together for some purpose and a timing issue arises. In situations like this, it can be very tedious and time-consuming to pour through the task list and cross-reference everything to make sure all the timing relationships are correct. So the Kit provides a convenient way to check timing information without having to wade through the task list.

From within any script, you can use the target identifier "state.timing" to retrieve the timing information for that script. The string returned has the format "phase/priority", consisting of the name of the phase followed by the numeric priority at which the script is running. By using this within a "debug" statement, you can report the timing of the current script. For example, you could use a statement like the one below to report the timing when a field is modified and its new value.

```
debug "Script 'blah' at " & state.timing & " - Field 'foo': " & field[foo].value
```

When you have numerous scripts all working together, or when you have the same field being setup by multiple different scripts under different conditions, this technique can be extremely handy. Along with the timing, you can include information about the script that is being invoked, and the net result is debug output that tells a very clear and specific story about what changes are occurring from which sources and at which times. Armed with detailed output like this, you should have a relative easy time nailing down the problem.

DIAGNOSING A TIMING ISSUE

Whenever you think you have a timing issue, the first diagnostic step is always to review each of the tasks involved in the behavior you're implementing. For each task, you need to double-check the associated tag expressions and/or script code to make sure everything should be working correctly.

Once you've verified that everything looks right, the next step is check the appropriate info windows to see whether all of your assumptions are being satisfied. Do fields contain the values they should? Are all the proper tags assigned where they should?

If everything appears correct, then you may very well have a timing issue. Why? Because the info windows only show what everything looks like at the **end** of the evaluation cycle. If TaskA is evaluated before TaskB, and TaskB sets a field value that TaskA depends upon, the field value will show the correct value within the info window. The problem in this case is that TaskA assumes the field will be set before it is invoked. The two tasks are being evaluated out of order, which can be corrected.

If everything is **not** correct within the info windows, then you may also have a timing issue. The key difference is that you also have some additional clues to help you isolate the problem this time. In a situation like this, the erroneous field values and/or tags will point you at the specific areas where the problem resides. If a field has the wrong value, then whatever scripts are setting that value are basing their logic on information that hasn't been setup properly yet or that has been further modified. If the tags are incorrect, then whatever scripts are assigning/deleting the tags are basing their logic on similarly wrong information. Keying on the extra clues should make the process of isolating the problem significantly quicker and easier.

ESTABLISHING TIMING DEPENDENCIES

Timing issues are a major nuisance when you are developing your data files. They are often hard to spot when you first introduce them, which means that you can't always assume that the problem arose with the last few changes that you made. Timing issues typically entail a tedious diagnostic process that takes the fun out of creating new data files. Most importantly, though, timing issues can quickly degenerate into a game of "Whack-A-Mole".

All of the different tasks within your data files are like a big chain of dominoes. They all need to be evaluated in the correct sequence to ensure that you reach the final goal of having an accurate character. However, if you move one domino, then that has ripple effects on the dominoes that were before and after it. Consequently, if you change the timing of one task to resolve a timing issue, you could find yourself simply creating a **different** timing issue between other tasks that have inter-dependencies with the task you moved. Squish one timing bug and up pops another.

Managing a large number of tasks and keeping all the timing dependencies clearly understood and enforced can quickly become a nightmare for an aspiring data file author. So the Kit provides a convenient mechanism that allows you to instantly detect when a timing issue has been introduced. We call the mechanism "timing dependencies" and it allows you to setup dependencies between different tasks that the compiler will verify for you automatically. This mechanism is the focus of the topics below.

NAMING TASKS

In order to establish a timing dependency on another task, you need to assign that task a name. By default, each task is assigned a name by the Kit, but it's not something you'll be able to establish a dependency upon. So any task that will be referenced within a dependency must be given a name. This is achieved by assigning the "name" attribute within the XML element that defines the task.

Task names can be just about anything you want. The goal is to identify what the task is doing clearly and succinctly, but you're the judge of how best to accomplish that. For example, you could name a task something detailed like "Calculate the Final Value for Attributes" or something brief like "Attr Final". The only criteria to keep in mind when choosing a name is that you'll need to type it anywhere that has a dependency.

ESTABLISHING "BEFORE" AND "AFTER" DEPENDENCIES

Named tasks become anchor points within the overall evaluation cycle for your data files. Other tasks will reference these anchors by establishing timing dependencies with them. This is accomplished by identifying the named task and specifying the nature of the dependency.

A dependency can be either a "before" or an "after" relationship. A "before" relationship tells the compiler that this task must always be performed before the named task. An "after" relationship indicates that the task must always be performed after the named task. Since all tasks are evaluated in a sequence, there is never a possibility that two tasks can be performed at the same time. Even if you assign them the identical phase and priority, the HL engine will assign an order to them. Consequently, each timing dependency must specify either a before or after relationship.

You can assign any number of timing dependencies to a given task. So you could define an assortment of dependencies that require a task to occur after certain tasks and an assortment that require that task to occur before others.

NOTE! Tasks do **not** need to be named in order to specify dependencies. Only tasks that serve as anchor points and are referenced by other tasks require names. Consequently, if you have a named task, you could have five different tasks depend on that task, and none of the dependent tasks require names.

USING THE TIMING REPORT

When your data files are compiled, a timing report is generated. You can view this timing report at any time by going to the "Debug" menu and selecting the "View Timing Report" option. For most users, your web browser will be launched and the file will be viewed within it. The timing report is an XML file that contains information about all of the timing dependencies you've defined and the tasks involved. Details on the contents will be found elsewhere within the Kit Reference documentation.

The most important sections in the timing report are the first three, so we'll discuss them here briefly. The first section identifies the names of any tasks that are unknown. These are names that are referenced by a timing dependency and that have not been defined. For example, if you specify that a task has a "before" dependency on the task named "My Task" and you have not assigned that name to any task, the name will be listed as unknown and no dependency will be established. You should always make sure that this section of the timing report is empty.

The second section identifies any names that have been duplicated and are not valid. It is perfectly legal to use the same name on multiple tasks. However, all tasks given the same name must be scheduled to occur at the exact same phase and priority. If you assign the name "My Task" to two separate tasks and they are scheduled at different times, they will be listed here as duplicates. You should always make sure that this section of the timing report is empty.

The third section identifies actual timing errors that exist between the tasks. If you specify that TaskA must occur before TaskB, and TaskA is assigned a phase and priority that is after TaskB, a timing error is reported and will be listed in this section. Any pair of tasks that does not satisfy the timing dependency assigned between them results in a timing error.

COMPILER CHECKING OF DEPENDENCIES

Whenever your data files are compiled, all of the timing dependencies are analyzed for you automatically. If the compiler identifies any errors in the timing dependencies you've specified, a corresponding error is reported. Unlike most errors, timing issues are treated merely as warning. This means that you are free to load the data files and use them, even though they will not work as you intended. The advantage of allowing you to load the files is that you can review both the timing report and the run-time information provided via info windows in an effort to resolve the problem.

With the timing dependency checks integrated into the compilation process, a critical safeguard is provided by HL. As you develop your files, you will add new logic and discover that the timing of some tasks needs to be adjusted in order to properly incorporate that new logic. If you change the timing of a task in order resolve a timing issue, you no longer have to worry about that change rippling into a different timing issue and having it go undetected. The compiler will use the timing dependencies to automatically detect whether the change has caused a ripple effect that also needs to be resolved. And the other sections of the timing report make it possible to pretty easily sort out even a chain of dependencies that become broken.

DECIDING WHAT DEPENDENCIES TO SETUP

After reading through all this, you might be thinking that you'll be spending a great deal of time setting up timing dependencies between all your tasks. That's not the case at all. In general, only a relatively small percentage of the tasks you define will have critical timing dependencies. Most tasks will be completely immune to dependency issues by simply assigning them to the appropriate phases. Consequently, you'll probably only find yourself needing to name maybe 10-20 tasks, and you'll likely only need to establish a similar number of dependencies upon those tasks.

The critical tasks that should be named as anchor points are those that satisfy three criteria. First, the task should be central to a series of evaluations that must occur in a specific order. Second, the tasks involved should be closely grouped within the evaluation cycle, such as all occurring with the same phase or a small number of sequential phases. Third, some of the tasks involved should have a reasonable chance of needing to be moved in the evaluation cycle as new logic is integrated into the data files.

If particular tasks satisfy all three of these criteria, then timing dependencies are extremely important. However, if only one or two of the criteria apply, then defining timing dependencies may or may not be justified. For example, if you have tasks that must setup values appropriately within fields before other tasks utilize those fields, setting up timing dependencies may be prudent. However, if the setup tasks are always performed within an early phase and the tasks using the values are always performed in a later phase, there is no need to establish timing dependencies.

SCRIPT CONTEXTS

This first step in accessing data via scripts is in identifying where that data resides within the overall data hierarchy. A separate hierarchy is maintained for both structural information (e.g. actors, picks, gizmos, minions, etc.) and visual information (e.g. panels, layouts, templates, etc.). Each different layer within the hierarchy is considered a distinct "context".

IMPORTANT! Within certain script contexts, all target references are treated as read-only, regardless of whether they are normally writable. Any attempts to modify the contents of a target reference that has been forced to be read-only will fail to either compile or work at run-time. For example, if a Position script for a visual element attempts to modify the contents of a pick, it will be forbidden. Scripts and/or contexts within which this behavior occurs should specify it within their documentation.

CONTEXTS WITHIN STRUCTURAL HIERARCHY

Within the structural hierarchy, there are a variety of contexts managed by the Kit. The following table identifies these contexts and provides a brief description of what each represents.

- **Portfolio:** The "portfolio" context represents the topmost level within the structural hierarchy, encapsulating all of the different leads that have been created. This context is generally not accessible via scripts, as each lead is considered to be an atomic object.
- **Hero:** The "hero" context represents an individual actor within the portfolio. This actor could be a lead or a minion.
- **Container:** The "container" context represents any container within the portfolio. The container could be an actor or a gizmo.
- **Pick:** The "pick" context represents any pick throughout the portfolio, located within any container.
- **Thing:** The "thing" context represents a thing that has not been added to the portfolio. The thing context is very similar to the pick context in behavior, with the only real difference being that the dynamic facets of picks don't exist for things, resulting in many valid actions for picks being inaccessible from things.
- **Field:** The "field" context represents any field within any pick or thing. If within a thing, all aspects of the field are read-only.
- **Pool:** The "pool" context represents any usage pool associated with either the actor or a specific pick.

There are also quite a few logical contexts within the structural hierarchy. Each of these contexts actually maps to one of the basic contexts above, but it is identified below due to important considerations, such as added restrictions or limitations on what can be done within the context.

- **Parent:** The "parent" context is wholly dependent on the current context. If the current context is a pick, then the parent is the container that the pick resides within. If the current context is a container, then the parent is the pick that attaches the gizmo, unless the container is an actor, in which case there is no parent.
- **Linkage:** The "linkage" context proceeds from one pick to another pick that has been associated with the first pick via a named linkage. If no linkage has been established, then there is no linkage context.
- **Root:** The "root" context identifies the pick that bootstrapped the current pick. If the pick was not bootstrapped by another pick, then there is no root context.
- **Dynalink:** The "dynalink" context provides access from one pick to another pick that is dynamically setup via the Creation script for the original pick. If no dynamic link is setup for a pick, there is no dynalink context.
- **Gearholder:** The "gearholder" context accesses the pick that is designated as the "holder" of the current pick. If the pick is not gear or is not presently assigned to another piece of gear as a holder, there is no gearholder context.
- **Shadow:** The "shadow" context represents the shadowed version of a pick, which resides within a different location in the structural hierarchy due the shadowing. If the pick is not shadowed, there is no shadow context.
- **Origin:** The "origin" context represents the original version of a displaced pick, which resides within a different location in the structural hierarchy due to the displacement. If the pick is not displaced, there is no origin context.
- **Master:** The "master" context identifies the actor that is the master of the current actor. If the current actor is not a minion, there is no master context.
- **Minion:** The "minion" context identifies an actor that is a minion of the current actor. If the current actor is not a master, there is no minion context.
- **Anchor:** The "anchor" context identifies the specific pick that attaches the current actor as a minion. If the current actor is not a minion, there is no anchor context.
- **Child:** The "child" context accesses a specific child pick that exists within the current container.
- **Gizmo:** The "gizmo" context references the gizmo that is attached as a child of the current pick. If the pick does not attach a gizmo, there is no gizmo context.

CONTEXTS WITHIN VISUAL HIERARCHY

The visual hierarchy has a separate set of contexts that are managed by the Kit and that are accessed exclusively via Position scripts on visual elements. The following table describes each of these contexts.

- **Scene:** The "scene" context identifies the top-level visual element within the current hierarchy, which will be either a panel, a form, or a sheet.
- **Layout:** The "layout" context identifies a layout within the top-level scene of the current hierarchy.
- **Template:** The "template" context identifies a template within the current hierarchy. Templates can either be used within layouts or within tables, so the parent context of the template within the hierarchy can vary.
- **Portal:** The "portal" context identifies a portal within the current hierarchy. Portals can either be used within templates or within layouts, so the parent context of the portal within the hierarchy can vary.
- **Table:** The "table" context identifies the containing table of a template or portal within the current hierarchy.
- **Value:** The "value" context represents any field within the pick or thing associated with a template. The value context is used for display only, so all aspects of the field are always read-only.

There are also a few logical contexts within the visual hierarchy. Each of these contexts actually maps to a basic context, but it is identified below due to important considerations, such as added restrictions or limitations on what can be done within the context.

- **Parent:** The "parent" context is wholly dependent on the current context, as detailed below.
  - *Portal:* The parent is the visual container that the portal resides within (i.e. a template or a layout).
  - *Template:* The parent is the visual container that the template resides within (i.e. a layout or a table).
  - *Layout:* The parent is the containing scene.
  - *Scene:* There is no parent for a scene.
- **Chosen:** The "chosen" context represents the current selection within a thing-based menu, so it can either be a pick or thing.
- **Hero:** The "hero" context within a visual script identifies the actor that the visual element is operating upon. This context is always treated as read-only from within a visual script.
- **Container:** The "container" context within a visual script identifies the container that the visual element is operating upon. This context is always treated as read-only from within a visual script.
- **Value:** The "value" context within a visual script represents the contents of a field. The field is accessed via a pick, which is identified by the template from which the value context is being utilized.

GENERAL CONTEXTS

In addition to the structural and visual contexts, the Kit supports contexts that are outside of the the normal hierarchy. These general contexts are described in the table below.

- **State:** The "state" context provides access to overall state information that pertains to the portfolio as a whole or general information about the evaluation cycle.
- **Transaction:** The "transaction" context represents the special pick that is utilized for buy and sell transactions. Since such transactions can be canceled by the user, all the details must be managed in a temporary fashion until the user completes the transaction.
- **Focus:** The "focus" context reflects the pick that has been established as the current "pick focus" via explicit actions within scripts. If no pick focus has been setup, then there is no focus context.
- **Actor:** The "actor" context reflects the hero that has been established as the current "actor focus" via explicit actions within scripts. If no actor focus has been setup, then there is no actor context.
- **Eachpick:** The "eachpick" context reflects the current pick that is being processed within a "foreach" loop performed by a script.
- **Altpick:** The "altpick" context identifies a secondary pick that is involved in an operation, such as when merging two stackable picks into one.
- **Altthing:** The "altthing" context identifies a secondary thing that is involved in an operation, such as is necessary when performing pre-requisite tests.

CONTEXTS DEFINED

HERO CONTEXT

The "hero" context represents an individual actor within the portfolio. This actor could be a lead or a minion.

CONTEXT TRANSITIONS

A "hero" context is very similar to a "container" context, except that the options available are a bit more limited (e.g. heroes don't have parents). From within a "hero" context, you can utilize the following set of valid context transitions:

Transition	Example	Definition
state	this.state	Transitions to the state context.
child[id]	this.child[mypick]	Transitions to the pick context corresponding to the first pick within the container that derives from the thing with the id specified. If the container has no child pick with the given unique id, a run-time error notification is reported to the user and the transition fails to resolve.
childfound[id]	this.childfound[mypick]	Transitions to the pick context corresponding to the first pick within the container that derives from the thing with the id specified. This transition is identical to "child[id]", except that the existence of the child pick is optional. If the child is found, the transition occurs normally. If the child does not exist, no run-time error is reported, although the transition still fails to resolve.

firstchild[expr ,sort]	this.firstchild[expr,mysort] this.firstchild[expr]	Transitions to the pick context corresponding to the first pick within the container that satisfies the tag expression given by expr . Since multiple children may satisfy the tag expression, an optional sort set id may be specified by sort, resulting in all matching children being sorted and the first child being used after the sort is performed. The tag expression may be either a literal string or a string expression. If no matching child exists, the transition fails to resolve.
minion[id]	this.minion[myminion]	Transitions to the hero context corresponding to the minion actor with the given id that exists beneath the actor that possesses the current container. If the container is not a master or the specified minion does not exist, the transition fails to resolve.
herofield[id]	this.herofield[myfield]	Transitions to the field context corresponding to the field given by id that exists on the "actor" pick for the containing actor. This is a shorthand notation for "hero.child[actor].field[id]".
usagepool[id]	this.usagepool[mypool]	Transitions to the pool context corresponding to the usage pool given by id that exists within the current actor.
transact	this.transact	Transitions to the pick context corresponding to the transaction pick that is associated with the hero governing the current context. **NOTE!** The transaction pick is only utilized within buy and sell transactions. As such, this transition is only valid within a few select scripts.
dynalink[index]	this.dynalink[myindex]	Transitions to the pick context corresponding to the registered dynamic linkage with the index specified. If no dynamic linkage has been registered with the given index, the transition fails to resolve. The index may be an arithmetic expression that calculates the actual index value to be used.

TARGET REFERENCES

Heroes are a special type of container. As such, they share all of the same target references as normal containers. However, they also have quite a few additional target references that are unique to heroes. The complete list of these special target references is presented in the table below.

IMPORTANT! For the purposes of data file authoring, the "hero" context applies to all actors, whether they be leads, masters, or minions.

IMPORTANT! Actors also support all general container target references.

NOTE! When an actor is accessed from within a visual script, all operations are restricted to read-only behavior. Any attempts to modify a container from within a visual script will fail.

Reference	Example	Definition
setactor	perform this.setactor	(Right, Number) Memorizes the current actor context as the "actor focus", allowing it to be instantly accessed thereafter via the "actor." initial script context. Always returns a value of zero.
miniontext	result = this.miniontext	(Right, String) Returns the name of the thing that attaches the current minion. This makes it possible to retrieve the nature of the minion for display to the user. If invoked on a lead, an empty string is returned.
sourcetree	this.sourcetree	(Right, String) Returns a summary of the various user-selected sources that have chosen for the current actor. This is intended for inclusion within character sheet output. The summary is synthesized solely from sources that are user-selected, although the entire chain of sources down to each selected source is included so that context is provided in case similar names are used in different contexts. Any source that is designated as not reportable is omitted from the summary, allowing sources governing printout and interface behaviors to be omitted. If a source specifies an alternate "reportname", that name is used instead.
errorcount	result = this.errorcount	(Right, Number) Returns the total number of validation errors that were reported within the hero over the course of evaluation. This is intended for use within character sheet output. If accessed during evaluation processing, a run-time error is reported.

errorlist	result = this.errorlist	(Right, String) Returns a string containing all validation errors for the hero, based on the most recent evaluation cycle. This is intended for use within character sheet output. If accessed during evaluation processing, a run-time error is reported.
combatant	result = this.combatant this.combatant = 0	(Right, Left, Number) Indicates whether the actor is managed as a combatant or not within the Tactical Console, with a non-zero value indicating a combatant. If a new value is assigned, the combatant state of the actor is changed.
initchange	result = this.initchange	(Right, Number) Returns non-zero if the actor's current initiative value has been modified by the user.
ismoveup	result = this.ismoveup	(Right, Number) Returns non-zero if the actor can be moved upwards within the Tactical Console, is thereby adjusting its initiative value.
ismovedown	result = this.ismovedown	(Right, Number) Returns non-zero if the actor can be moved downwards within the Tactical Console, thereby adjusting its initiative value.
combatmove[expr]	perform this.combatmove[-1]	(Right, Number) Moves the actor upwards or downwards within the Tactical Console, thereby adjusting its initiative position. The number of slots to move is given by the numeric expression expr, with a positive value moving the actor later in the initiative sequence and a negative value earlier. To move an actor down in the order (i.e. later in combat), use "+1", and use "-1" to move the actor upwards. To move an actor to the top or bottom of the initiative order, simply use a very large value, as the engine will safely bound all adjustments. Always returns a value of zero.
combatact	perform this.combatact	(Right, Number) Designates the actor as having acted this combat turn and triggers an update to determine the next "ready" actor(s). Always returns a value of zero.
combatdefer	perform this.combatdefer	(Right, Number) Designates the actor as having deferred its action this combat turn and triggers an update to determine the next "ready" actor(s). Always returns a value of zero.
combatmovenow	perform this.combatmovenow	(Right, Number) Moves the actor within the initiative sequence such that it is positioned immediately before the first actor that is currently designated as "ready". This is intended for use when a deferred actor finally acts, as its initiative position is moved to the current slot in the queue. Always returns a value of zero.

CONTAINER CONTEXT

The "container" context represents any container within the portfolio. The container could be an actor or a gizmo.

IMPORTANT! If the container context happens to be an actor, then you can utilize the context as either a standard container context or as a Hero Context.

CONTEXT TRANSITIONS

From within a "container" context, you can utilize the following set of valid context transitions:

Transition	Example	Definition
state	this.state	Transitions to the state context.
hero	this.hero	Transitions to the hero context corresponding to the hero that is the parent of the current container. If the current container is a hero, then this transition changes nothing but does resolve successfully.
container	this.container	Transitions to the container context corresponding to whatever container is the immediate parent of the current container. If the current container is a hero, then the transition fails to resolve.
parent	this.parent	Transitions to the pick context corresponding to the parent pick that attaches the container. If the parent container is a hero and has no parent pick, the transition fails to resolve.

child[id]	this.child[mypick]	Transitions to the pick context corresponding to the first pick within the container that derives from the thing with the id specified. If the container has no child pick with the given unique id, a run-time error notification is reported to the user and the transition fails to resolve.
childfound[id]	this.childfound[mypick]	Transitions to the pick context corresponding to the first pick within the container that derives from the thing with the id specified. This transition is identical to "child[id]", except that the existence of the child pick is optional. If the child is found, the transition occurs normally. If the child does not exist, no run-time error is reported, although the transition still fails to resolve.
firstchild[expr,sort]	this.firstchild[expr,mysort] this.firstchild[expr]	Transitions to the pick context corresponding to the first pick within the container that satisfies the tag expression given by expr. Since multiple children may satisfy the tag expression, an optional sort set id may be specified by sort, resulting in all matching children being sorted and the first child being used after the sort is performed. The tag expression may be either a literal string or a string expression. If no matching child exists, the transition fails to resolve.
anchor	this.anchor	Transitions to the pick context corresponding to the pick within the master actor that attaches the current actor as a minion. If the container does not reside within a minion, the transition fails to resolve.
master	this.master	Transitions to the hero context corresponding to the master actor for which this container is a minion. If the container is not a minion, the transition fails to resolve.
minion[id]	this.minion[myminion]	Transitions to the hero context corresponding to the minion actor with the given id that exists beneath the actor that possesses the current container. If the container is not a master or the specified minion does not exist, the transition fails to resolve.
herofield[id]	this.herofield[myfield]	Transitions to the field context corresponding to the field given by id that exists on the "actor" pick for the containing actor. This is a shorthand notation for "hero.child[actor].field[id]".
useagepool[id]	this.usagepool[mypool]	Transitions to the pool context corresponding to the usage pool given by id that exists within the current container. This transition is only valid for actors, since gizmos do not possess usage pools.
transact	this.transact	Transitions to the pick context corresponding to the transaction pick that is associated with the hero governing the current context. **NOTE!** The transaction pick is only utilized within buy and sell transactions. As such, this transition is only valid within a few select scripts.
dynalink[index]	this.dynalink[myindex]	Transitions to the pick context corresponding to the registered dynamic linkage with the index specified. If no dynamic linkage has been registered with the given index, the transition fails to resolve. The index may be an arithmetic expression that calculates the actual index value to be used.

TARGET REFERENCES

There are a wide assortment of operations that can be performed on containers, some of which modify the container and some that simply retrieve information about the container. The container context applies equally to heroes and gizmos, although there are some behavioral differences that arise for some target references. The complete list of target references for containers is presented in the table below.

NOTE! When a container is accessed from within a visual script, all operations are restricted to read-only behavior. Any attempts to modify a container from within a visual script will fail.

Reference	Example	Definition

live	result = this.live	(Right, Number) Returns non-zero if the container is currently considered "live" else zero if non-live. The live state for a gizmo is dictated by the live state of its parent pick that attaches it, while a hero is always considered live.
ishero	result = this.ishero	(Right, Number) Returns non-zero if the container is a hero, else zero for a gizmo.
isactive	result = this.isactive	(Right, Number) Returns non-zero if the container either is or resides within the currently active actor within the HL interface. If the container is or resides within a different actor, zero is returned.
livename	result = this.livename	(Right, String) Returns a suitable name for the container that is based on dynamic changes made via scripts. If the container is an actor, the name of the actor is returned, else the name of the parent pick of the gizmo is returned.
actorname	result = this.actorname	Returns the name of the actor that encompasses the current container context. The name is whatever has been assigned by the user.
playername	result = this.playername	(Right, String) Returns the name of the player that is associated with the current lead. The name is whatever has been entered by the user.
assign[tag]	perform this.assign[skill.appraise]	(Right, Number) Assigns the indicated tag to the container. The tag must be specified using the standard "group.id" syntax. The tag is verified to exist during compilation of the script. Always returns a value of zero.
delete[tmpl]	perform this.delete[skill.craft?]	(Right, Number) Deletes all tags from the container that match the tag template tmpl. The template must use the standard "group.id" syntax and may contain a wildcard. If the template employs a wildcard, all tags matching the template are deleted. If the template specifies an explicit tag, and the tag has been assigned to the container multiple times, the tag is deleted only once, thereby providing detailed control to authors when needed. Always returns a value of zero.
assignstr[str]	perform this.assignstr[skill.appraise]	(Right, Number) This target reference is identical to "assign", except that the str parameter is a string expression that is evaluated within the script. This allows the tag to be dynamically determined via the script instead of being hard-wired at compilation. Always returns a value of zero.
deletestr[str]	perform this.deletestr["skill.craft?"]	(Right, Number) This target reference is identical to "delete", except that the str parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation. Always returns a value of zero.
tagis[tmpl]	this.tagis[skill.?]	(Right, Number) Returns non-zero if any tags assigned to the container match the tag template tmpl , else zero if no tags match. The template must use the standard "group.id" syntax and may contain a wildcard.

tagcount[tmpl]	result = this.tagcount[skill.?]	(Right, Number) Returns the number of tags assigned to the container that match the tag template tmpl . The template must use the standard "group.id" syntax and may contain a wildcard.
tagvalue[tmpl]	result = this.tagvalue[spelllevel.wizard?]	(Right, Number) Returns the value of a tag assigned to the container that matches the tag template tmpl . The rules associated with tag values in tag expressions apply. The template must use the standard "group.id" syntax and may contain a wildcard.
tagmin[tmpl]	this.tagmin[spelllevel.wizard?]	(Right, Number) Returns the minimum value of all tags assigned to the container that match the tag template tmpl . The rules associated with tag values in tag expressions apply. The template must use the standard "group.id" syntax and may contain a wildcard.
tagmax[tmpl]	result = this.tagmax[spelllevel.wizard?]	(Right, Number) Returns the maximum value of all tags assigned to the container that match the tag template tmpl . The rules associated with tag values in tag expressions apply. The template must use the standard "group.id" syntax and may contain a wildcard.
tagunique[tmpl]	result = this.tagunique[skill.?]	(Right, Number) Returns the number of unique tags assigned to the container that match the tag template tmpl . If a tag is assigned multiple times, it is only counted once. The template must use the standard "group.id" syntax and may contain a wildcard.
tagexpr[expr]	result = this.tagexpr[class.wizard & (val:spelllevel.wizard? > 5)]	(Right, Number) Returns non-zero if the tags assigned to the container match the tag expression expr , else zero is returned. The expr parameter must be a valid tag expression.
tagcountstr[str]	this.tagcountstr["skill.?"]	(Right, Number) This target reference is identical to "tagcount", except that the str parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation.
tagvaluestr[str]	result = this.tagvaluestr["spelllevel.wizard?"]	(Right, Number) This target reference is identical to "tagvalue", except that the str parameter is a string expression that is evaluated within the script. This allows the tag template to bedynamically determined via the script instead of being hard-wired at compilation.
tagminstr[str]	result = this.tagminstr["spelllevel.wizard?"]	(Right, Number) This target reference is identical to "tagmin", except that the str parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation.
tagmaxstr[str]	result = this.tagmaxstr["spelllevel.wizard?"]	(Right, Number) This target reference is identical to "tagmax", except that the str parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation.

taguniquestr[str]	result = this.taguniquestr["skill.?"]	(Right, Number) This target reference is identical to "tagunique", except that the str parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation.
tagsearch[str]	result = this.tagsearch["class.wizard & (val:spelllevel.wizard? > 5)"]	(Right, Number) This target reference is identical to "tagexpr", except that the str parameter is a string expression that is evaluated within the script. This allows the tag expression to be dynamically determined via the script instead of being hard-wired at compilation. **NOTE!** Using "tagsearch" is MUCH slower in performance than using "tagexpr", since the tagexpr must be parsed and compiled every time the script is evaluated. Be sure to use "tagexpr" whenever possible.
tagmatch[refgrp,match,ref]	result = hero.tagmatch[NeedStatus,HasStatus,initial]	(Right, Number) Returns non-zero if any tags within a reference context also exist within a match context, else zero. This allows tags from one context to be verified to exist within another context, making tag-based comparisons between two objects possible. For more details, please check here.
heromatch[refgrp,match,ref]	result = this.heromatch[NeedStatus,HasStatus,initial]	(Right, Number) Returns non-zero if any tags within the actor also exist within a match context, else zero. This allows tags from the actor to be verified to exist within another context, making tag-based comparisons between the actor and another object possible. The key difference between "heromatch" and "tagmatch" is that this target reference always assumes the initial context is the actor, which is then compared against the context that is transitioned to. For more details, please check here.
intersect[init,curr]	result = this.interset[MyGroup,AltGroup]	(Right, Number) Returns non-zero if tags within the initial script context also exist within the currently transitioned context, else zero. All of the tags that belong to the init tag group within the initial script context are compared against the tags that belong to the curr tag group within the current context. If any of those tags possess the identical tag id, then a match is returned (i.e. non-zero). The same tag group may be used for both contexts, and both contexts must be objects that possess tags (i.e. container, pick, or thing).
inherit[id]	result = this.inherit[mypick result = this.inherit	(Right, Number) The container inherits all tags from one or all of its child picks. If an id is given, then all tags from the pick with that id are inherited into the container. If no parameter is given, then all tags from all child picks are inherited. The return value is the total number of tags inherited.

pulltags[tmpl,grp]	result = hero.pulltags[thingid.?] result = hero.pulltags[thingid.?,Ability]	(Right, Number) Copies tags from the transitioned container context into the initial pick or container context and returns the total number of tags copied. The set of tags to be copied from the transitioned context are dictated by the tag template tmpl . If the second parameter is omitted, the identified tags are copied to the initial context. If the second parameter is provided, grp must specify a tag group. All tags identified by tmpl are mapped to equivalent tags within the tag group grp , and those mapped tags are then assigned to the initial context. When mapping, an equivalent mapped tag must exist for all identified tags. Both the initial and transitioned script context must be either picks or containers.
pushtags[tmpl,grp]	result = hero.pushtags[thingid.?] result = hero.pushtags[thingid.?,Ability]	(Right, Number) Copies tags from the initial pick or container context into the transitioned container context and returns the total number of tags copied. This target reference is identical in behavior to "pulltags", except that the tags are copied in the opposite direction.
childexists[id]	result = this.childexists[mypick]	(Right, Number) Returns non-zero if any child pick with the given id exists within the container, else zero if no matching pick is found.
childlives[id]	result = this.childlives[mypick]	(Right, Number) Returns non-zero if any child pick with the given id exists within the container and is "live". If either no matching pick is found or all matching picks found are "non-live", zero is returned.
childcount[id]	result = this.childcount[mypick]	(Right, Number) Returns the number of child picks with the given id that exist within the container. A value of zero indicates that no matching picks were found.
haschild[str]	result = this.haschild["component.Skill"]	(Right, Number) Returns the number of child picks within the container that match the tag expression given by str. The parameter is a string expression that must contain a valid tag expression and is tested against all children of the container. The parameter can be synthesized dynamically within the script.
setidentity[grp]	result = this.setidentity[groupid]	(Right, Number) Assigns to the container the identity tag from the tag group grp that corresponds to the initial context of the script. The identity tag id is dictated by the initial context of the script. For more details, please check here.
isidentity[grp]	result = this.isidentity[groupid]	(Right, Number) As the counterpart of "setidentity", this target reference returns non-zero if the indicated identity tag has been assigned to the container and zero if not. The identity sought must be from the tag group grp and the tag id is dictated by the initial context of the script. For more details, please check here.

countidentity[grp]	result = this.countidentity[groupid]	(Right, Number) Similar to "isidentity", this target reference returns the count of the identity tags assigned to the container. The identity tag sought must be from the tag group grp and the tag id is dictated by the initial context of the script. For more details, please check here.
isminion	result = this.isminion	(Right, Number) Returns non-zero if the container is or resides within a minion, else zero.
ismaster	result = this.ismaster	(Right, Number) Returns non-zero if the container is or resides within a master, else zero.
hasminion[id]	result = this.hasminion[myminion]	(Right, Number) Returns non-zero if the container is or resides within an actor that possesses a minion with the specified id. If the container is not within a master or does not contain the specified minion, zero is returned.
isdynalink[expr]	result = this.isdynalink[4]	(Right, Number) Returns non-zero if a dynamic linkage has been defined for the implied hero context with the index specified, else zero if no linkage exists. If the container is a gizmo, then the containing actor is used as the hero context. The value given by index may be an arithmetic expression that will be resolved properly at run-time.
istransact	result = this.istransact	(Right, Number) Returns non-zero if a viable transaction context exists for the container. This allows scripts to check that a transaction context exists before attempting to transition into the transaction context.
panelvalid[id]	result = this.panelvalid[mypanel]	(Left, Number) Sets the validity state of the tab panel given by id. If the value assigned is zero, the panel is designated as non-valid and its name will be highlighted in red to the user. Since the default state of all panels is valid, if a non-zero value is assigned, this target reference is ignored. This means you can simply designate a panel as invalid when appropriate and do nothing when the panel is valid. **NOTE!** This target reference can only be used from within an Eval Rule or a Validate script.
tagnames[tmpl,spl]	result = this.tagnames[Weapon.?,"+"]	(Right, String) Generates and returns a list of tags within the container. Only tags that match the tag template tmpl are included in the list. The generated string appends the names of the tags together, inserting the splicing string spl between each name if there is more than one. A container with no tags matching the template will return the empty string.
tagabbrevs[tmpl,spl]	result = this.tagabbrevs[Weapon.?,"+"]	(Right, String) Works identically to "tagnames", except that the generated string is comprised of the abbreviations for all matching tags instead of their names.
tagids[tmpl,spl]	result = this.tagids[Weapon.?,"+"]	(Right, String) Works identically to "tagnames", except that the generated string is comprised ofthe unique ids for all matching tags instead of their names.

weight	result = this.weight	(Right, Number) Returns the total weight of all "gear" picks within the container.
gearlist[spl,expr]	result = this.gearlist["+",!Equipment.Natural]	(Right, String) Generates and returns a list of gear held within the container, which means gear that is **not** held within another gear holder. Only picks that are designated as gear and that are not assigned to a holder are candidates for inclusion in the list. Each candidate piece of gear is compared against the tag expression expr, and only those that satisfy the tag expression are included in the final list. The generated string appends the names of all pieces of gear together, inserting the splicing string spl between each name if there is more than one. A container that holds no gear matching the tag expression will return the empty string.
geartree[expr]	result = this.geartree[!Helper.NoMove]	(Right, String) Generates and returns a hierarchical tree view of the gear picks possessed by the container context, as appropriate for use within the Dashboard and Tactical Console. Only picks that are designated as gear are candidates for inclusion in the report. Each candidate piece of gear is compared against the tag expression expr, and only those that satisfy the tag expression are included in the final report. All gear is output in a hierarchy, where each level of containment is indented beneath the level above it. Gear within child gizmos is also included, with indentation as if the gizmo were a holder within the container and its contents indented beneath it.

PICK CONTEXT

The "pick" context represents any pick throughout the portfolio, located within any container.

CONTEXT TRANSITIONS

From within a "pick" context, you can utilize the following set of valid context transitions:

Transition	Example	Definition
state	this.state	Transitions to the state context.
hero	this.hero	Transitions to the hero context corresponding to the hero that contains the current pick.
container	this.container	Transitions to the container context corresponding to the immediate container of the current pick, whether it be a hero or a gizmo.
parent	this.parent	Transitions to the container context corresponding to the immediate container of the current pick, just like the "container" transition.
gizmo	this.gizmo	Transitions to the container context corresponding to the child gizmo directly attached by the pick. If the pick has no attached child gizmo, the transition fails to resolve.
field[id]	this.field[myfield]	Transitions to the field context corresponding to the field within the current pick that has the id specified. If the given field does not exist for the current pick, the transition fails to resolve. **NOTE!** Within things and picks, there are a number of pre-defined pseudo-fields that are always defined and that allow access to facets of the pick that are not governed by user-defined fields. These pseudo-fields behave like normal fields in all respects within scripts, except that some are read-only. The list of pre-defined fields can be found elsewhere in the Kit Reference documentation.

root	this.source	Transitions to the pick context corresponding to the root pick that bootstraps the current pick into the container. If the current pick is not bootstrapped, or if the current pick is designated as unique and can possess multiple bootstraps, the transition fails to resolve.
gearholder	this.gearholder	Transitions to the pick context corresponding to the pick that is assigned as the gear holder of the current pick. If the current pick is not held, the transition fails to resolve. If the current pick is not gear, an error is reported and the transition fails to resolve.
linkage[id]	this.linkage[mylink]	Transitions to the pick context corresponding to the linkage with the id specified. If the linkage is not defined, an error is reported.
anchor	this.anchor	Transitions to the pick context corresponding to the pick within the master actor that attaches the current actor as a minion. If the pick does not reside within a minion, the transition fails to resolve.
master	this.master	Transitions to the hero context corresponding to the master actor for which this pick's container is a minion. If the pick is not within a minion, the transition fails to resolve.
minion[id]	this.minion[myminion] this.minion	Transitions to the hero context corresponding to the minion actor with the given id that exists beneath the actor that possesses the current pick. The id can be omitted, in which case the minion is implicitly identified and must be directly attached by the current pick. If the pick does not reside within a master or the specified minion does not exist, the transition fails to resolve.
herofield[id]	this.herofield[myfield]	Transitions to the field context corresponding to the field given by id that exists on the "actor" pick for the containing actor. This is a shorthand notation for "hero.child[actor].field[id]".
usagepool[id]	this.usagepool[mypool]	Transitions to the pool context corresponding to the usage pool with the id specified for the current pick.
shadow	this.shadow	Transitions to the container context corresponding to the container into which the current pick is shadowed. If the pick is not shadowed, an error is reported and the transition fails to resolve.
origin	this.origin	Transitions to the container context corresponding to the container into which the current pick was originally added. If the pick is displaced, the container is where the pick was added by the user. If not displaced, the container is simply the container for the pick.

TARGET REFERENCES

Picks are going to be the most prevalent object type within any set of data files, so it's no surprise that picks have the largest and most varied assortment of target references. The complete list of target references for picks is presented in the table below.

NOTE! When a pick is accessed from within a visual script, all operations are restricted to read-only behavior. Any attempts to modify a pick from within a visual script will fail.

Reference	Example	Definition
livename	result = this.livename	(Right, String) Returns an appropriate name for the pick. If the user has explicitly named the pick, that name is returned. If not named by the user, any name change dictated by scripts is returned. Otherwise, the name of the thing is used.
idstring	result = this.idstring	(Right, String) Returns the unique id of the thing as a string. This can be extremely handy when synthesizing tag templates and tag expressions on-the-fly via scripts.
valid	result = this.valid	(Right, Number) Returns non-zero if the pick is valid and zero if the pick has been designated as non-valid. Validity is controlled via mechanisms like pre-requisite tests and Eval Rules.

isuser	result = this.isuser	(Right, Number) Returns non-zero if the pick was directly added by the user, else zero. Picks that are bootstrapped by containers or other picks are not considered user-added, even if the pick that does the bootstrapping is user-added.
ispick	result = this.ispick	(Right, Number) Returns non-zero if the current context is a pick and zero if it's a thing. This provides a means to discern the nature of the context when the circumstances make a distinction uncertain, such as within procedures.
isroot	result = this.isroot	(Right, Number) Returns non-zero if the pick has been bootstrapped and therefore has a root pick available.
isgizmo	result = this.isgizmo	(Right, Number) Returns non-zero if the pick directly attaches a child gizmo.
isentity	result = this.isentity	(Right, Number) Returns non-zero if the pick directly attaches a child entity. **NOTE!** This target reference is essentially identical to "isgizmo" (above).
isunique	this.isunique	(Right, Number) Returns non-zero if the pick behaves as unique.
shadowed	result = this.shadowed	(Right, Number) Returns non-zero if the pick has been shadowed and also exists within an alternate container.
displaced	result = this.displaced	(Right, Number) Returns non-zero if the pick has been displaced and also exists within an alternate container.
countme	result = this.countme	(Right, Number) Returns the total number of instances of the current pick that exist within the container of the current pick. As long as the thing id matches, two picks are considered equivalent. If stacking is utilized, this target reference returns the total number of distinct picks within the container, ignoring all stacked quantities.
uniqcount	result = this.uniqcount	(Right, Number) Return the number of times that a unique pick has been added to the current container. Every time a unique pick is added, whether by the user or via bootstrapping, it's reference count is incremented, and this target reference returns the reference count. If the pick is not unique, a run-time error is reported and zero is returned.
assign[tag]	perform this.assign[skill.appraise]	(Right, Number) Assigns the indicated tag to the pick. The tag must be specified using the standard "group.id" syntax. The tag is verified to exist during compilation of the script. Always returns a value of zero.

delete[tmpl]	perform this.delete[skill.craft?]	(Right, Number) Deletes all tags from the pick that match the tag template tmpl. The template must use the standard "group.id" syntax and may contain a wildcard. If the template employs a wildcard, all tags matching the template are deleted. If the template specifies an explicit tag, and the tag has been assigned to the pick multiple times, the tag is deleted only once, thereby providing detailed control to authors when needed. Always returns a value of zero.
assignstr[str]	perform this.assignstr[skill.appraise]	(Right, Number) This target reference is identical to "assign", except that the str parameter is a string expression that is evaluated within the script. This allows the tag to be dynamically determined via the script instead of being hard-wired at compilation. Always returns a value of zero.
deletestr[str]	perform this.deletestr["skill.craft?"]	(Right, Number) This target reference is identical to "delete", except that the str parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation. Always returns a value of zero.
tagis[tmpl]	this.tagis[skill.?]	(Right, Number) Returns non-zero if any tags assigned to the pick match the tag template tmpl, else zero if no tags match. The template must use the standard "group.id" syntax and may contain a wildcard.
tagcount[tmpl]	result = this.tagcount[skill.?]	(Right, Number) Returns the number of tags assigned to the pick that match the tag template tmpl . The template must use the standard "group.id" syntax and may contain a wildcard.
tagvalue[tmpl]	result = this.tagvalue[spelllevel.wizard?]	(Right, Number) Returns the value of a tag assigned to the pick that matches the tag template tmpl . The rules associated with tag values in tag expressions apply. The template must use the standard "group.id" syntax and may contain a wildcard.
tagmin[tmpl]	result = this.tagmin[spelllevel.wizard?]	(Right, Number) Returns the minimum value of all tags assigned to the pick that match the tag template tmpl. The rules associated with tag values in tag expressions apply. The template must use the standard "group.id" syntax and may contain a wildcard.
tagmax[tmpl]	result = this.tagmax[spelllevel.wizard?]	(Right, Number) Returns the maximum value of all tags assigned to the pick that match the tag template tmpl. The rules associated with tag values in tag expressions apply. The template must use the standard "group.id" syntax and may contain a wildcard.
tagunique[tmpl]	result = this.tagunique[skill.?]	(Right, Number) Returns the number of unique tags assigned to the pick that match the tag template tmpl . If a tag is assigned multiple times, it is only counted once. The template must use the standard "group.id" syntax and may contain a wildcard.

tagexpr[expr]	result = this.tagexpr[class.wizard & (val:spelllevel.wizard? > 5)]	(Right, Number) Returns non-zero if the tags assigned to the pick match the tag expression expr , else zero is returned. The expr parameter must be a valid tag expression.
tagcountstr[str]	result = this.tagcountstr["skill.?"]	(Right, Number) This target reference is identical to "tagcount", except that the str parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation.
tagvaluestr[str]	result = this.tagvaluestr["spelllevel.wizard?"]	(Right, Number) This target reference is identical to "tagvalue", except that the str parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation.
tagminstr[str]	result = this.tagminstr["spelllevel.wizard?"]	(Right, Number) This target reference is identical to "tagmin", except that the str parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation.
tagmaxstr[str]	result = this.tagmaxstr["spelllevel.wizard?"]	(Right, Number) This target reference is identical to "tagmax", except that the str parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation.
taguniquestr[str]	result = this.taguniquestr["skill.?"]	(Right, Number) This target reference is identical to "tagunique", except that the str parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation.
tagsearch[str]	result = this.tagsearch["class.wizard & (val:spelllevel.wizard? > 5)"]	(Right, Number) This target reference is identical to "tagexpr", except that the str parameter is a string expression that is evaluated within the script. This allows the tag expression to be dynamically determined via the script instead of being hard-wired at compilation. **NOTE!** Using "tagsearch" is MUCH slower in performance than using "tagexpr", since the tagexpr must be parsed and compiled every time the script is evaluated. Be sure to use "tagexpr" whenever possible.
tagmatch[refgrp,match,ref]	result = hero.tagmatch[NeedStatus,HasStatus,initial]	(Right, Number) Returns non-zero if any tags within a reference context also exist within a match context, else zero. This allows tags from one context to be verified to exist within another context, making tag-based comparisons between two objects possible. For more details, please check here.

heromatch[refgrp,match,ref]	result = hero.heromatch[NeedStatus,HasStatus,initial]	(Right, Number) Returns non-zero if any tags within the actor also exist within a match context, else zero. This allows tags from the actor to be verified to exist within another context, making tag-based comparisons between the actor and another object possible. The key difference between "heromatch" and "tagmatch" is that this target reference always assumes the initial context is the actor, which is then compared against the context that is transitioned to. For more details, please check here.
intersect[init,curr]	result = this.interset[MyGroup,AltGroup]	(Right, Number) Returns non-zero if tags within the initial script context also exist within the currently transitioned context, else zero. All of the tags that belong to the init tag group within the initial script context are compared against the tags that belong to the curr tag group within the current context. If any of those tags possess the identical tag id, then a match is returned (i.e. non-zero). The same tag group may be used for both contexts, and both contexts must be objects that possess tags (i.e. container, pick, or thing).
pulltags[tmpl,grp]	result = this.pulltags[thingid.?] result = this.pulltags[thingid.?,Ability]	(Right, Number) Copies tags from the transitioned pick context into the initial pick or container context and returns the total number of tags copied. The set of tags to be copied from the transitioned context are dictated by the tag template tmpl . If the second parameter is omitted, the identified tags are copied to the initial context. If the second parameter is provided, grp must specify a tag group. All tags identified by tmpl are mapped to equivalent tags within the tag group grp , and those mapped tags are then assigned to the initial context. When mapping, an equivalent mapped tag must exist for all identified tags. Both the initial and transitioned script context must be either picks or containers.
pushtags[tmpl,grp]	result = this.pushtags[thingid.?] result = this.pushtags[thingid.?,Ability]	(Right, Number) Copies tags from the initial pick or container context into the transitioned pick context and returns the total number of tags copied. This target reference is identical in behavior to "pulltags", except that the tags are copied in the opposite direction.

forward[tmpl,target]	result = this.forward[thingid.?,shadow] result = this.forward[thingid.?] result = this.forward	(Right, Number) Copies tags from the pick context into the corresponding container and returns the total number of tags copied. The set of tags to be copied are dictated by the tag template tmpl . The tags are copied to the container dictated by target, which must be one of the following values: • *parent:* Tags are copied to the standard parent container of the pick. • *shadow:* Tags are copied to the shadow container of the pick, which is only applicable when the pick has been shadowed. • *origin:* Tags are copied to the origin container of the pick, which is only applicable when the pick has been displaced. The target parameter may be omitted, in which case all tags matching the tag template are copied to the "parent" container. Both parameters may also be omitted, in which case all tags within the pick are copied to the "parent" container.
setidentity[grp]	result = this.setidentity[groupid]	(Right, Number) Assigns to the pick the identity tag from the tag group grp that corresponds to the initial context of the script. The identity tag id is dictated by the initial context of the script. For more details, please check here.
isidentity[grp]	result = this.isidentity[groupid]	(Right, Number) As the counterpart of "setidentity", this target reference returns non-zero if the indicated identity tag has been assigned to the pick and zero if not. The identity tag sought must be from the tag group grp and the tag id is dictated by the initial context of the script. For more details, please check here.
countidentity[grp]	result = this.countidentity[groupid]	(Right, Number) Similar to "isidentity", this target reference returns the count of the identity tags assigned to the pick. The identity tag sought must be from the tag group grp and the tag id is dictated by the initial context of the script. For more details, please check here.
shareidentity[grp,pick]	perform this.shareidentity[ClassSkill,mypick]	(Right, Number) The pick with unique id pick is assigned the identity tag from the tag group grp for the current pick context. This allows a script to explicitly identify a pick and assign an identity tag to it. The identity tag id is dictated by the initial context of the script. For more details, please check here. If the specified pick does not exist, a run-time error is reported. A value of zero is always returned.
pullidentity[grp]	perform this.pullidentity[groupid]	(Right, Number) Locates the identity tag from the tag group grp for the current pick context and assigns it to the initial script context. The identity tag id is dictated by the initial context of the script. For more details, please check here. If the initial script context is neither a pick nor a container, a run-time error is reported. A value of zero is always returned.

tagnames[tmpl,spl]	result = this.tagnames[Weapon.?,"+"]	(Right, String) Generates and returns a list of tags within the pick. Only tags that match the tag template tmpl are included in the list. The generated string appends the names of the tags together, inserting the splicing string spl between each name if there is more than one. A pick with no tags matching the template will return the empty string.
tagabbrevs[tmpl,spl]	result = this.tagabbrevs[Weapon.?,"+"]	(Right, String) Works identically to "tagnames", except that the generated string is comprised of the abbreviations for all matching tags instead of their names.
tagids[tmpl,spl]	result = this.tagids[Weapon.?,"+"]	(Right, String) Works identically to "tagnames", except that the generated string is comprised of the unique ids for all matching tags instead of their names.
prereqok	result = this.prereqok	(Right, Number) Returns non-zero if the pick satisfies all of its pre-requisites. May not be used during evaluation.
prereqnum	result = this.prereqnum	(Right, Number) Returns the total number of pre-requisite rules that exist for the current pick. May not be used during evaluation.
prereqmsg[index]	result = this.prereqmsg[i]	(Right, String) Returns the message associated with a specific pre-requisite test for the current pick, where the desired test is given by index. The index is zero-based, so the values 0 through 4 should be used for a pick with 5 pre-requisites. If the individual pre-requisite is satisfied, the text returned is the empty string. Otherwise, the text returned is the corresponding message. May not be used during evaluation. **NOTE!** Retrieving the message for individual pre-requisite tests will trigger a re-calculation every time, so it is expensive and should only be used sparingly.
isgear	result = this.isgear	(Right, Number) Returns non-zero if the pick is a piece of gear.
isholdable	result = this.isholdable	(Right, Number) Returns non-zero if the pick is gear that can be placed into a gear holder.
isgearheld	result = this.isgearheld	(Right, Number) Returns non-zero if the pick is a piece of gear that is currently being held within another piece of gear.
gearcount	result = this.gearcount	(Right, Number) Returns the total number of pieces of gear that are currently being held within the pick. If the pick is not a gear holder, the count will always be zero.
gearpath[spl]	result = this.gearpath["+"]	(Right, String) Generates and returns the entire gear containment hierarchy for this piece of gear. The hierarchy starts with the topmost gear holder of this pick and works downward through the immediate holder of the pick, showing the sequence. The generated string lists all the holders by name, inserting the splicing string spl between each name if there is more than one. A piece of gear that is not held will return the empty string.

isgearlist	result = this.isgearlist	(Right, Number) Returns non-zero if the pick is a gear holder and can contain a list of held gear, regardless of whether any gear is actually held.
gearlist[spl,expr]	result = this.gearlist["+",!Equipment.Natural]	(Right, String) Generates and returns a list of gear held within the pick, provided the pick is a gear holder. Only picks that are designated as gear and that are assigned to this pick as their holder are candidates for inclusion in the list. Each candidate piece of gear is compared against the tag expression expr , and only those that satisfy the tag expression are included in the final list. The generated string appends the names of all pieces of gear together, inserting the splicing string spl between each name if there is more than one. A pick that holds no gear matching the tag expression will return the empty string.
stackable	result = this.stackable	(Right, Number) Returns non-zero if the pick can be stacked with other equivalent picks.
isbootstrap[thing]	this.isbootstrap[thingid]	(Right, Number) Returns non-zero if the current pick directly bootstraps a pick with the specified id thing.
rootnames[spl]	result = this.rootnames["+"]	(Right, String) Generates and returns a list of picks that bootstrap the current pick. The generated string appends the names of the root picks together, inserting the splicing string spl between each name if there is more than one. A pick with no root picks will return the empty string.
islinkage[link]	result = this.islinkage[linkid]	(Right, Number) Returns non-zero if the current pick possesses a linkage with the specified id link.
autonomous	result = this.autonomous	(Right, Number) Returns non-zero if the pick has zero other picks that are currently based upon it. Picks that have other picks based upon them are typically made non-deletable, so this autonomous target reference provides a means to detect that condition.
creation	result = this.creation	(Right, Number) Returns non-zero if the pick was added to the character during "creation" mode and zero if added during "advancement" mode. This makes it possible to validate options that are only seelctable during character creation. If advancement mode is not enabled for the game system, all picks are always added in creation mode.
setfocus	perform this.setfocus	(Right, Number) Establishes the current pick context as the new "pick focus". Thereafter, the script can use the "focus." initial context transition to directly access the pick that is setup as the focus. Always returns a value of zero.
ispanel	result = this.ispanel	(Right, Number) Returns non-zero is the pick has a panel linkage defined for it. This allows generic handling of panel linkages within component scripts when the actual panel linkage is optionally controlled by the thing.

panelactive	result = this.panelactive	(Right, Number) Returns non-zero if the current active tab panel is the one specified as a panel linkage for the current pick. If the pick has no designated panel linkage, zero is returned.
sourcerefs[spl]	result = this.sourcerefs["+"]	(Right, String) Generates and returns a list of all sources that the current pick is dependent upon. The generated string appends the names of the sources together, inserting the splicing string spl between each name if there is more than one. A pick with no source dependencies will return the empty string.
uniqindex	result = this.uniqindex	(Right, Number) Returns an integer value that uniquely identifies the pick throughout the entire portfolio. This value is *not* guaranteed to be the same when the portfolio is reloaded. It is intended for use in differentiating picks within rules and should never be saved in any way or otherwise used as a persistent reference.
dynareg[index]	perform this.dynareg[42]	(Right, Number) Registers a dynamic linkage to the current pick context and assigns that linkage the unique index value index. Once registered, the current pick can be accessed from the "hero" context via the "dynalink" context transition. This makes it possible to dynamically setup access to a special pick throughout an actor. This target reference can only be utilized from within a Creation Script. Always returns a value of zero.
isminion	result = this.isminion	(Right, Number) Returns non-zero if the pick resides within a minion, else zero.
ismaster	result = this.ismaster	(Right, Number) Returns non-zero if the pick resides within a master, else zero.
hasminion[id]	result = this.hasminion[myminion] result = this.hasminion	(Right, Number) Returns non-zero if the pick resides within an actor that possesses a minion with the specified id. The parameter can be omitted, in which case non-zero is returned only if the pick directly attaches a minion. If the pick is not within a master or the specified minion does not exist, zero is returned.

THING CONTEXT

The "thing" context represents a thing that has not been added to the portfolio. The thing context is very similar to the pick context in behavior, with the only real difference being that the dynamic facets of picks don't exist for things, resulting in many valid actions for picks being inaccessible from things.

CONTEXT TRANSITIONS

A "thing" context is very similar to a "pick" context, except that a "thing" context represents a thing that has not yet been added to a container and therefore lacks any dynamic state. As a result, the "thing" context is much more restrictive than the "pick" context. From within a "thing" context, you can utilize the following set of valid context transitions:

Transition	Example	Definition
state	this.state	Transitions to the state context.
field[id]	this.field[myfield]	Transitions to the field context corresponding to the field within the current thing that has the id specified. If the given field does not exist for the current thing, the transition fails to resolve. **NOTE!** Within things and picks, there are a number of pre-defined pseudo-fields that are always defined and that allow access to facets of the pick that are not governed by user-defined fields. These pseudo-fields

		behave like normal fields in all respects within scripts, except that some are read-only. The list of pre-defined fields can be found elsewhere in the Kit Reference documentation.
linkage[id]	this.linkage[mylink]	Transitions to the pick context corresponding to the linkage with the id specified. If the linkage is not defined, an error is reported.

TARGET REFERENCES

There are a number of target references that apply to things that have not been added to a container. These come into play at times like testing pre-requisites and when things are selected via menus. While similar to picks in many ways, things have no dynamic facets and are therefore always read-only in behavior. The complete list of target references for thing is presented in the table below.

Reference	Example	Definition
idstring	result = this.idstring	(Right, String) Returns the unique id of the thing as a string. This can be extremely handy when synthesizing tag templates and tag expressions on-the-fly via scripts.
ispick	result = this.ispick	(Right, Number) Returns non-zero if the current context is a pick and zero if it's a thing. This provides a means to discern the nature of the context when the circumstances make a distinction uncertain, such as within procedures.
isunique	result = this.isunique	(Right, Number) Returns non-zero if the thing behaves as unique.
isentity	result = this.isentity	(Right, Number) Returns non-zero if the thing directly attaches a child entity.
tagis[tmpl]	this.tagis[skill.?]	(Right, Number) Returns non-zero if any tags assigned to the thing match the tag template tmpl, else zero if no tags match. The template must use the standard "group.id" syntax and may contain a wildcard.
tagcount[tmpl]	result = this.tagcount[skill.?]	(Right, Number) Returns the number of tags assigned to the thing that match the tag template tmpl. The template must use the standard "group.id" syntax and may contain a wildcard.
tagvalue[tmpl]	result = this.tagvalue[spelllevel.wizard?]	(Right, Number) Returns the value of a tag assigned to the thing that matches the tag template tmpl. The rules associated with tag values in tag expressions apply. The template must use the standard "group.id" syntax and may contain a wildcard.
tagmin[tmpl]	result = this.tagmin[spelllevel.wizard?]	(Right, Number) Returns the minimum value of all tags assigned to the thing that match the tag template tmpl. The rules associated with tag values in tag expressions apply. The template must use the standard "group.id" syntax and may contain a wildcard.
tagmax[tmpl]	result = this.tagmax[spelllevel.wizard?]	(Right, Number) Returns the maximum value of all tags assigned to the thing that match the tag template tmpl. The rules associated with tag values in tag expressions apply. The template must use the standard "group.id" syntax and may contain a wildcard.
tagunique[tmpl]	result = this.tagunique[skill.?]	(Right, Number) Returns the number of unique tags assigned to the thing that match the tag template tmpl. If a tag is assigned multiple times,

		it is only counted once. The template must use the standard "group.id" syntax and may contain a wildcard.
tagexpr[expr]	result = this.tagexpr[class.wizard & (val:spelllevel.wizard? > 5)]	(Right, Number) Returns non-zero if the tags assigned to the thing match the tag expression expr , else zero is returned. The expr parameter must be a valid tag expression.
tagcountstr[str]	result = this.tagcountstr["skill.?"]	(Right, Number) This target reference is identical to "tagcount", except that the str parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation.
tagvaluestr[str]	this.tagvaluestr["spelllevel.wizard?"]	(Right, Number) This target reference is identical to "tagvalue", except that the str parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation.
tagminstr[str]	this.tagminstr["spelllevel.wizard?"]	(Right, Number) This target reference is identical to "tagmin", except that the str parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation.
tagmaxstr[str]	result = this.tagmaxstr["spelllevel.wizard?"]	(Right, Number) This target reference is identical to "tagmax", except that the str parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation.
taguniquestr[str]	result = this.taguniquestr["skill.?"]	(Right, Number) This target reference is identical to "tagunique", except that the str parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation.
tagsearch[str]	result = this.tagsearch["class.wizard & (val:spelllevel.wizard? > 5)"]	(Right, Number) This target reference is identical to "tagexpr", except that the str parameter is a string expression that is evaluated within the script. This allows the tag expression to be dynamically determined via the script instead of being hard-wired at compilation. **NOTE!** Using "tagsearch" is MUCH slower in performance than using "tagexpr", since the tagexpr must be parsed and compiled every time the script is evaluated. Be sure to use "tagexpr" whenever possible.
tagmatch[refgrp,match,ref]	result = hero.tagmatch[NeedStatus,HasStatus,initial]	(Right, Number) Returns non-zero if any tags within a reference context also exist within a match context, else zero. This allows tags from one context to be verified to exist within another context, making tag-based comparisons between two objects possible. For more details, please check here.
intersect[init,curr]	result = this.interset[MyGroup,AltGroup]	(Right, Number) Returns non-zero if tags within the initial script context also exist within the

		currently transitioned context, else zero. All of the tags that belong to the init tag group within the initial script context are compared against the tags that belong to the curr tag group within the current context. If any of those tags possess the identical tag id, then a match is returned (i.e. non-zero). The same tag group may be used for both contexts, and both contexts must be objects that possess tags (i.e. container, pick, or thing).
tagnames[tmpl,spl]	result = this.tagnames[Weapon.?,"+"]	(Right, String) Generates and returns a list of tags within the thing. Only tags that match the tag template tmpl are included in the list. The generated string appends the names of the tags together, inserting the splicing string spl between each name if there is more than one. A thing with no tags matching the template will return the empty string.
tagabbrevs[tmpl,spl]	result = this.tagabbrevs[Weapon.?,"+"]	(Right, String) Works identically to "tagnames", except that the generated string is comprised of the abbreviations for all matching tags instead of their names.
tagids[tmpl,spl]	result = this.tagids[Weapon.?,"+"]	(Right, String) Works identically to "tagnames", except that the generated string is comprised of the unique ids for all matching tags instead of their names.
prereqok	result = this.prereqok	(Right, Number) Returns non-zero if the pick satisfies all of its pre-requisites. May not be used during evaluation.
prereqnum	result = this.prereqnum	(Right, Number) Returns the total number of pre-requisite rules that exist for the current pick. May not be used during evaluation.
prereqmsg[index]	result = this.prereqmsg[i]	(Right, String) Returns the message associated with a specific pre-requisite test for the current pick, where the desired test is given by index. The index is zero-based, so the values 0 through 4 should be used for a pick with 5 pre-requisites. If the individual pre-requisite is satisfied, the text returned is the empty string. Otherwise, the text returned is the corresponding message. May not be used during evaluation. **NOTE!** Retrieving the message for individual pre-requisite tests will trigger a re-calculation every time, so it is expensive and should only be used sparingly.
isgear	result = this.isgear	(Right, Number) Returns non-zero if the thing is a piece of gear.
isholdable	result = this.isholdable	(Right, Number) Returns non-zero if the pick is gear that can be placed into a gear holder.
stackable	result = this.stackable	(Right, Number) Returns non-zero if the thing can be stacked with other equivalent picks.
isbootstrap[thing]	result = this.isbootstrap[thingid]	(Right, Number) Returns non-zero if the current thing directly bootstraps a thing with the specified id thing.
islinkage[link]	result = this.islinkage[linkid]	(Right, Number) Returns non-zero if the current thing possesses a linkage with the specified id link.

ispanel	result = this.ispanel	(Right, Number) Returns non-zero if the thing has a panel linkage defined for it. This allows generic handling of panel linkages within component scripts when the actual panel linkage is optionally controlled by the thing.
sourcerefs[spl]	this.sourcerefs["+"]	(Right, String) Generates and returns a list of all sources that the current thing is dependent upon. The generated string appends the names of the sources together, inserting the splicing string spl between each name if there is more than one. A thing with no source dependencies will return the empty string.
amendthing[targ ,str]	result = this.amendthing[name,"New Name"]	(Right, String) Modifies the name or description of the thing henceforth for the current actor. Once amended, all subsequent references to the thing will return the new contents. This includes the details shown within chooser tables that display things for selection by the user. The targ parameter must be either "name" or "description", specifying which facet of the thing is being amended. The new contents are given by the string expression str. Always returns a value of zero. **NOTE!** The amendment is applied to the thing at the time of evaluation, so any access of the name or description on derived picks prior to the amendment will not show the effects of the amendment. All amendments persist until the start of the next evaluation cycle, at which time they are all reset. **NOTE!** Each actor maintains its own set of amendments, so it is perfectly valid to have multiple different actors with different amendments to the same thing.

FIELD CONTEXT

The "field" context represents any field within any pick or thing. If within a thing, all aspects of the field are read-only.

CONTEXT TRANSITIONS

From within a "field" context, you can utilize the following set of valid context transitions:

Transition	Example	Definition
state	this.state	Transitions to the state context.
pick	this.pick	Transitions to the pick context corresponding to the pick that contains the current field.
chosen	this.chosen	Transitions to either the pick context corresponding to the user-selected pick stored within the field or the thing context corresponding to the user-selected thing stored within the field, depending on the nature of the field. The transition is only valid for use on fields that store menu selections made by the user.

TARGET REFERENCES

There are a variety of ways to access and manipulate the contents of fields. The complete list of target references for fields is presented in the table below.

NOTE! When a field is accessed from within a thing or a visual script, all operations are restricted to read-only behavior. Any attempts to modify a field from within a thing context or via a visual script will fail.

Reference	Example	Definition
value	this.field[myfield].value this.field[myfield].value = 42	(Left, Right, Number) Accesses the contents of the field as a numeric value. If the field is text and the contents retrieved, the contents are automatically converted to a suitable value.
text	result = this.field[myfield].text this.field[myfield].text = "hello"	(Left, Right, String) Accesses the contents of the field as a string. If the field is numeric can the contents retrieved, the contents are automatically converted to a suitable string.
isempty	result = this.field[myfield].isempty	(Right, Number) Returns non-zero if the field contains the empty string and zero if it contains text of any length. If used on a numeric field or with an array or matrix, an error is reported. Testing of array and matrix elements must be done via the "compare" intrinsic.
ischanged	result = this.field[myfield].ischanged	(Right, Number) Returns non-zero if the field contents have been changed in any way from its original starting state. This can detect a field that has been changed by the user or via a script.
reset	perform this.field[myfield].reset	(Right, Number) The contents of the field are reset to the initial default value for the thing. A value of zero is always returned.
ischosen	result = this.field[myfield].ischosen	(Right, Number) Returns non-zero if the field contains a pick or a thing that was selected via a menu. If the field is associated with a menu and no selection has yet been made, zero is returned.
delta	result = this.field[myfield].delta this.field[myfield].delta = 2	(Left, Right, Number) Accesses the delta component of a user field for manipulation separately from the user-specified value. The field must be explicitly configured to support delta processing.
arrayvalue[row]	result = this.field[myfield].arrayvalue[3] this.field[myfield].arrayvalue[3] = 42	(Left, Right, Number) Accesses the contents of a specific element of the array-based field as a numeric value. The index of the element is given by row, where the index is a zero-based value that must be less than the number of rows in the array. If the field is text and the contents retrieved, the contents are automatically converted to a suitable value.
arraytext[row]	result = this.field[myfield].arraytext[3] this.field[myfield].arraytext[3] = "hello"	(Left, Right, Number) Accesses the contents of a specific element of the array-based field as a string. The index of the element is given by row, where the index is a zero-based value that must be less than the number of rows in the array. If the field is numeric can the contents retrieved, the contents are automatically converted to a suitable string.
matrixvalue[row,col]	result = this.field[myfield].matrixvalue[3,4] this.field[myfield].matrixvalue[3,4] = 42	(Left, Right, Number) Accesses the contents of a specific element of the matrix-based field as a numeric value. The index of the element is given by row and col, where the indices are zero-based values that must be less than the number of rows and columns in the matrix, respectively. If the field is text and the contents retrieved, the contents are automatically converted to a suitable value.

matrixtext[row,col]	result = this.field[myfield].matrixvalue[3,4] this.field[myfield].matrixvalue[3,4] = 42	(Left, Right, String) Accesses the contents of a specific element of the matrix-based field as a string. The index of the element is given by row and col, where the indices are zero-based values that must be less than the number of rows and columns in the matrix, respectively. If the field is numeric and the contents retrieved, the contents are automatically converted to a suitable string.
arraydump	result = this.field[myfield].arraydump	(Right, String) Generates and returns a text string that contains the values of all elements in the array, with the results appropriately formatted for easy viewing. This mechanism is ideal for use within "debug" statements to help isolate scripting problems with arrays. Only usable with array-based fields. **NOTE!** The generated output is capped to the limits of the debug output mechanism, so using this mechanism with large arrays and/or text-based fields may result in the output being truncated.
matrixdump	result = this.field[myfield].matrixdump	(Right, String) Generates and returns a text string that contains the values of all elements in the matrix, with the results appropriately formatted for easy viewing. This mechanism is ideal for use within "debug" statements to help isolate scripting problems with matrices. Only usable with matrix-based fields. **NOTE!** The generated output is capped to the limits of the debug output mechanism, so using this mechanism with large matrices and/or text-based fields may result in the output being truncated.
datetime[fmt,sep]	result = this.field[myfield].datetime[gamedate,"/"]	(Right, String) Returns the contents of the field formatted for output as a date or time, where sep is the separator string to use between values. The fmt parameter dictates the formatting to be used and must be one of the following values: • *realdate:* Formatted as if it's a real-world date. • *realtime:* Formatted as if it's a real-world time. • *gamedate:* Formatted as if it's a game-world date. • *gametime:* Formatted as if it's a game-world time. **NOTE!** This target reference is only supported on fields within picks - not things.
modify[oper,val,str]	perform this.field[myfield].modify[+,1,"first"]	(Right, Number) Applies a modification to the field with accompanying notes for tracking a history of changes. The modification nature is given by the oper parameter, which which specifies the operation to be performed. The val parameter is an arithmetic expression the provides the modification value to be applied. The str parameter is a text annotation that is recorded along with the modification. If str is the empty string, the notes are the name of the initial pick

		context that is applying the change (or "??? ??" if the initial context is not a pick). The oper parameter must be one of the following: • '+' - The value is added to the field. • '-' - The value is subtracted from the field. • '*' - The value is multiplied into the field. • '/' - The value is divided into the field. • '=' - The new value is assigned to the field and all previous adjustments to the field are ignored. This operator is only valid with "stack" or "changes" history tracking. • '#' - The value is added to the field, but no sign is displayed for the field within the history report. This allows the identification of basic values included into the field calculation that are distinct from bonuses (which include the '+'). • '$' - The value is completely ignored and only the text entry is added for the modification. This allows a history entry to be recorded that has no value but that needs to be included in the report. The field must be explicitly configured to support history tracking. Always returns a value of zero.
history[spl,start]	result = this.field[myfield].history result = this.field[myfield].history["&"] result = this.field[myfield].history[", ",42]	(Right, String) Generates and returns a text string that contains the change history for the field. The spl parameter is a string that is inserted between entries in the change history, splicing them together for appropriate output. The start parameter is the starting value used for the field within the report. Both the spl and start parameters are optional, although the spl is required in order to specify the start parameter. If omitted, the spl parameter defaults to a comma and a space (", "), while the start parameter defaults to zero. The history report contents depend on the history behavior assigned to the field, as given below: • ***best***: Only the notes text for each history entry is reported • ***stack:*** The notes text for each entry is reported and the adjustment details are included in parentheses with the notes (e.g. "notes (+2)") • ***changes:*** Same as 'stack'

POOL CONTEXT

The "pool" context represents any usage pool associated with either the actor or a specific pick.

CONTEXT TRANSITIONS

From within a "pool" context, you can utilize the following set of valid context transitions:

Transition	Example	Definition
-None-	N/A	There are no transitions from within a pool context.

TARGET REFERENCES

The "pool" script context corresponds to the contents of a usage pool, whether it be associated with a pick or directly on the actor. The complete list of target references for usage pools is presented in the table below.

Reference	Example	Definition
name	result = this.name	(Right, String) Returns the name assigned to the usage pool.
abbrev	result = this.abbrev	(Right, String) Returns the abbreviation assigned to the usage pool.
count	result = this.count	(Right, Number) Returns the number of historical entries that currently exist within the usage pool.
value	result = this.value	(Right, Number) Returns the net adjusted value for the usage pool, after applying all of the adjustments that have been assigned.
empty	perform this.empty	(Right, Number) Discards all adjustments for the usage pool and resets the value to its initial default. Always returns a value of zero.
adjust[value]	perform this.adjust[42]	(Right, Number) Applies an adjustment to the usage pool in the amount given by the value parameter, which can be an arithmetic expression of any type. Always returns a value of zero.
set[value]	perform this.set[42]	(Right, Number) Applies an adjustment to the usage pool that sets the net value of the pool to the new total given by the value parameter. For example, if the current net value of the pool is 14 and value is 17, an adjustment of 3 is applied. Always returns a value of zero.
rollback	perform this.rollback	(Right, Number) Rolls back (i.e. undoes) the most recent adjustment that was applied to the usage pool. Always returns a value of zero.
history[index]	result = this.history[3]	(Right, Number) Returns the value of an individual adjustment in the change history. The specific entry is given by the index parameter, which can be an arithmetic expression. The index is zero-based, so the index must a value between zero and the total number of entries in the usage pool minus one. The zeroth entry is the most recent adjustment, with an increasing value proceeding further backwards in history. This provides a way to retrieve a report of all of the adjustments in the history of the pool and format them however you want.

SCENE CONTEXT

The "scene" context identifies the top-level visual element within the current hierarchy, which will be either a panel, a form, or a sheet.

CONTEXT TRANSITIONS

From within a "scene" context, you can utilize the following set of valid context transitions:

Transition	Example	Definition
layout[id]	this.layout[mylayout]	Transitions to the layout context corresponding to the layout within the current scene that possesses the id specified. If the layout does not exist within the scene, the transition fails to resolve.
container	this.container	Transitions to the container context corresponding to the container to which the scene applies, whether it be an actor or a gizmo. **NOTE!** After transitioning, access within the new container context will

		be read-only and limited in what information can be retrieved. **NOTE!** This transition can only be used as the first transition when within a visual script.
hero	this.hero	Transitions to the hero context corresponding to the hero to which the scene applies. **NOTE!** After transitioning, access within the new hero context will be read-only and limited in what information can be retrieved. **NOTE!** This transition can only be used as the first transition when within a visual script.

TARGET REFERENCES

The "scene" script context applies equally to panels, forms, and sheets. However, there are some important behavioral differences between those three visual elements that impact how certain target references operate for each, and those differences are detailed below. The complete list of target references for scenes is presented in the table below.

Reference	Example	Definition
width	result = this.width this.width = 420	(Left, Right, Number) Accesses the width of the scene. The width of panels and sheets is properly setup by HL and any changes are completely ignored, as HL wholly controls the rendering region for panels and sheets. For forms, the width is initialized to something safe by HL, but the author is assumed to set the width appropriately for the contents that need to be displayed.
height	result = this.height this.height = 420	(Left, Right, Number) Accesses the height of the scene. The same rules apply as for the "width" target reference above.
scrollbar	result = this.scrollbar	(Right, Number) Returns the width of a scroller, in pixels.
defwidth	result = this.defwidth	(Right, Number) Returns the default width for a form that is specified within the form's definition. Only applicable to forms.
defheight	result = this.defheight	(Right, Number) Returns the default height for a form that is specified within the form's definition. Only applicable to forms.
minwidth	result = this.minwidth this.minwidth = 420	(Left, Right, Number) Accesses the minimum width governing the form. A value of zero indicates the minimum width should be the default width. If both minwidth and maxwidth are zero, the form cannot be resized by the user. If data files specify both minwidth and maxwidth as zero, the form width cannot be modified via scripts. If the min/max values are modified such that the current dimensions of the form become invalid, the form dimensions are automatically adjusted to comply with the new limits. Only applicable to forms.
minheight	result = this.minheight this.minheight = 420	(Left, Right, Number) Accesses the minimum height governing the form. The same rules apply as for the "minwidth" target reference above.
maxwidth	result = this.maxwidth this.maxwidth = 420	(Left, Right, Number) Accesses the maximum width governing the form. The same rules apply as for the "minwidth" target reference above.

maxheight	result = this.maxheight this.maxheight = 420	(Left, Right, Number) Accesses the maximum height governing the form. The same rules apply as for the "minwidth" target reference above.

LAYOUT CONTEXT

The "layout" context identifies a layout within the top-level scene of the current hierarchy.

CONTEXT TRANSITIONS

From within a "layout" context, you can utilize the following set of valid context transitions:

Transition	Example	Definition
parent	this.parent	Transitions to the scene context corresponding to the visual element that contains the layout.
template[id]	this.template[mytemplate]	Transitions to the template context corresponding to the template within the current layout that possesses the id specified. If the template does not exist within the layout, the transition fails to resolve. **NOTE!** The id specified is the id of the template reference, as defined within the layout, and not necessarily the id of the template itself.
portal[id]	this.portal[myportal]	Transitions to the portal context corresponding to the portal within the current layout that possesses the id specified. If the portal does not exist within the layout, the transition fails to resolve. **NOTE!** The id specified is the id of the portal reference, as defined within the layout, and not necessarily the id of the portal itself. **NOTE!** Only portals defined directly within the layout can be accessed via this transition. Portals within templates must be accessed via the template.
container	this.container	Transitions to the container context corresponding to the container to which the layout applies, whether it be an actor or a gizmo. **NOTE!** After transitioning, access within the new container context will be read-only and limited in what information can be retrieved. **NOTE!** This transition can only be used as the first transition when within a visual script.
hero	this.hero	Transitions to the hero context corresponding to the hero to which the layout applies. **NOTE!** After transitioning, access within the new hero context will be read-only and limited in what information can be retrieved. **NOTE!** This transition can only be used as the first transition when within a visual script.

TARGET REFERENCES

The "layout" script context governs the operations that can be applied to layouts within scenes. The complete list of target references for layouts is presented in the table below.

Reference	Example	Definition
width	result = this.width this.width = 420	(Left, Right, Number) Accesses the width of the layout. Unless explicitly specified within the XML, the width of a layout is initialized to the width of the containing scene, minus any assigned margins.
height	result = this.height this.height = 420	(Left, Right, Number) Accesses the height of the scene. Unless explicitly specified within the XML, the height of a layout is initialized to the height of the containing scene, minus any assigned margins.

Reference	Example	Definition
left	result = this.left this.left = 42	(Left, Right, Number) Accesses the position of the left edge of the layout within the containing visual element.
top	result = this.top this.top = 42	(Left, Right, Number) Accesses the position of the top edge of the layout within the containing visual element.
right	result = this.right	(Right, Number) Returns the position of the right edge of the layout within the containing visual element.
bottom	result = this.bottom	(Right, Number) Returns the position of the bottom edge of the layout within the containing visual element.
visible	result = this.visible this.visible = 1	(Left, Right, Number) Controls the visibility of the layout within the containing visual element. A non-zero value indicates the layout is visible and a zero value indicates hidden.
scrollbar	result = this.scrollbar	(Right, Number) Returns the width of a scroller, in pixels.
autotop	result = this.autotop this.autotop = 42	(Left, Right, Number) Accesses the position of the top edge of the auto-place region within the containing visual element.
autobottom	result = this.autobottom this.autobottom = 420	(Left, Right, Number) Accesses the position of the bottom edge of the auto-place region within the containing visual element.
autoleft	result = this.autoleft this.autoleft = 42	(Left, Right, Number) Accesses the position of the left edge of the auto-place region within the containing visual element.
autoright	result = this.autoright this.autoright = 420	(Left, Right, Number) Accesses the position of the right edge of the auto-place region within the containing visual element.
autowidth	result = this.autowidth this.autowidth = 420	(Left, Right, Number) Accesses the width of the auto-place region within the containing visual element.
autoheight	result = this.autoheight this.autoheight = 420	(Left, Right, Number) Accesses the height of the auto-place region within the containing visual element.
autogap	result = this.autogap this.autogap = 42	(Left, Right, Number) Accesses the default gap size used when automatically placing elements within the containing visual element. The "autogap" defaults to zero.
autoplace[gap]	perform this.autoplace[42] perform this.autoplace	(Right, Number) Automatically places the layout within the containing visual element, subject to the standard rules for automatic placement. The gap parameter specifies the gap to be used between this layout and the previous placed element. The parameter can be omitted, in which case the established "autogap" is utilized. A value of zero is always returned.

TEMPLATE CONTEXT

The "template" context identifies a template within the current hierarchy. Templates can either be used within layouts or within tables, so the parent context of the template within the hierarchy can vary.

CONTEXT TRANSITIONS

From within a "template" context, you can utilize the following set of valid context transitions:

Transition	Example	Definition
parent	this.parent	Transitions to the layout context or table context corresponding to the visual element that contains the tmeplate. **NOTE!** The "parent" transition can only be utilized a single time, so it is not possible to go upwards two or more levels within the hierarchy.
portal[id]	this.portal[myportal]	Transitions to the portal context corresponding to the portal within the current template that possesses the id specified. If the portal does not exist within the template, the transition fails to resolve. **NOTE!** Only portals defined directly within the template can be accessed via this transition.
field[id]	this.field[myfield]	Transitions to the value context corresponding to the field within the current template that has the id specified. The fields for a template are dictated by the pick or thing that is associated with the template. If the given field does not exist for the pick/thing, the transition fails to resolve. **NOTE!** Within templates, all fields are treated as read-only, which is controlled by transitioning to a distinct "value" context instead of "field" context.
container	this.container	Transitions to the container context corresponding to the container to which the template applies, whether it be an actor or a gizmo. **NOTE!** After transitioning, access within the new container context will be read-only and limited in what information can be retrieved. **NOTE!** This transition can only be used as the first transition when within a visual script.
hero	this.hero	Transitions to the hero context corresponding to the hero to which the template applies. **NOTE!** After transitioning, access within the new hero context will be read-only and limited in what information can be retrieved. **NOTE!** This transition can only be used as the first transition when within a visual script.

TARGET REFERENCES

The "template" script context governs the operations that can be applied to templates within layouts and tables. The complete list of target references for templates is presented in the table below.

Reference	Example	Definition
width	result = this.width this.width = 420	(Left, Right, Number) Accesses the width of the template. Unless explicitly specified within the XML, the width of a template is automatically initialized. If the template is within a table, the width is set the interior width of the containing table, minus any assigned margins. Otherwise, the template is initialized to a width of 100.
height	result = this.height this.height = 420	(Left, Right, Number) Accesses the height of the template. Unless explicitly specified within the XML, the height of a layout is automatically initialized to zero and must be set by the author via a suitable script.
left	result = this.left this.left = 42	(Left, Right, Number) Accesses the position of the left edge of the template within the containing visual element.

Reference	Example	Definition
top	result = this.top this.top = 42	(Left, Right, Number) Accesses the position of the top edge of the template within the containing visual element.
right	result = this.right	(Right, Number) Returns the position of the right edge of the template within the containing visual element.
bottom	result = this.bottom	(Right, Number) Returns the position of the bottom edge of the template within the containing visual bottom element.
visible	result = this.visible this.visible = 1	(Left, Right, Number) Controls the visibility of the template within the containing visual element. A non-zero value indicates the template is visible and a zero value indicates hidden. The visibility of templates cannot be controlled within tables.
isroot	result = this.isroot	(Right, Number) Returns non-zero if the pick associated with the template has been bootstrapped and therefore has a root pick available. If the pick is associated with a thing, zero is returned.
issizing	result = this.issizing	(Right, Number) Returns non-zero if the "sizing" logic of a template within a table is being performed. This allows detection of the sizing logic so that all non-sizing behaviors can be skipped for the template.
inheader	result = this.inheader	(Right, Number) Returns non-zero if the template is being positioned for use as a header within a table. This makes it possible to distinguish the context for a template that is being used as a dual-purpose header.
scrollbar	result = this.scrollbar	(Right, Number) Returns the width of a scroller, in pixels.
tagis[tmpl]	this.tagis[skill.?]	(Right, Number) Returns non-zero if any tags assigned to the pick/thing associated with the template that match the tag template tmpl, else zero if no tags match. The tag template must use the standard "group.id" syntax and may contain a wildcard.
tagcount[tmpl]	result = this.tagcount[skill.?]	(Right, Number) Returns the number of tags assigned to the pick/thing associated with the template that match the tag template tmpl. The tag template must use the standard "group.id" syntax and may contain a wildcard.
tagvalue[tmpl]	result = this.tagvalue[spelllevel.wizard?]	(Right, Number) Returns the value of a tag assigned to the pick/thing associated with the template that matches the tag template tmpl. The rules associated with tag values in tag expressions apply. The tag template must use the standard "group.id" syntax and may contain a wildcard.

Reference	Example	Definition
tagmin[tmpl]	result = this.tagmin[spelllevel.wizard?]	(Right, Number) Returns the minimum value of all tags assigned to the pick/thing associated with the template that match the tag template tmpl. The rules associated with tag values in tag expressions apply. The tag template must use the standard "group.id" syntax and may contain a wildcard.
tagmax[tmpl]	result = this.tagmax[spelllevel.wizard?]	(Right, Number) Returns the maximum value of all tags assigned to the pick/thing associated with the template that match the tag template tmpl. The rules associated with tag values in tag expressions apply. The tag template must use the standard "group.id" syntax and may contain a wildcard.
tagunique[tmpl]	result = this.tagunique[skill.?]	(Right, Number) Returns the number of unique tags assigned to the pick/thing associated with the template that match the tag template tmpl. If a tag is assigned multiple times, it is only counted once. The tag template must use the standard "group.id" syntax and may contain a wildcard.
tagexpr[expr]	result = this.tagexpr[class.wizard & (val:spelllevel.wizard? > 5)]	(Right, Number) Returns non-zero if the tags assigned to the pick/thing associated with the template match the tag expression expr , else zero is returned. The expr parameter must be a valid tag expression.
tagcountstr[str]	result = this.tagcountstr["skill.?"]	(Right, Number) This target reference is identical to "tagcount", except that the str parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation.
tagvaluestr[str]	result = this.tagvaluestr["spelllevel.wizard?"]	(Right, Number) This target reference is identical to "tagvalue", except that the str parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation.
tagminstr[str]	result = this.tagminstr["spelllevel.wizard?"]	(Right, Number) This target reference is identical to "tagmin", except that the str parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation.
tagmaxstr[str]	result = this.tagmaxstr["spelllevel.wizard?"]	(Right, Number) This target reference is identical to "tagmax", except that the str parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation.
taguniquestr[str]	result = this.taguniquestr["skill.?"]	(Right, Number) This target reference is identical to "tagunique", except that the str parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation.

Reference	Example	Definition
isidentity[grp]	result = this.isidentity[groupid]	Right, Number) Returns non-zero if the indicated identity tag has been assigned to the pick/thing associated with the template. The identity tag sought must be from the tag group grp and the tag id is dictated by the initial context of the script. For more details, please check here.
autotop	result = this.autotop this.autotop = 42	(Left, Right, Number) Accesses the position of the top edge of the auto-place region within the containing visual element.
autobottom	result = this.autobottom this.autobottom = 420	(Left, Right, Number) Accesses the position of the bottom edge of the auto-place region within the containing visual element.
autoleft	result = this.autoleft this.autoleft = 42	(Left, Right, Number) Accesses the position of the left edge of the auto-place region within the containing visual element.
autoright	result = this.autoright this.autoright = 420	(Left, Right, Number) Accesses the position of the right edge of the auto-place region within the containing visual element.
autowidth	result = this.autowidth this.autowidth = 420	(Left, Right, Number) Accesses the width of the auto-place region within the containing visual element.
autoheight	result = this.autoheight this.autoheight = 420	(Left, Right, Number) Accesses the height of the auto-place region within the containing visual element.
autogap	result = this.autogap this.autogap = 42	(Left, Right, Number) Accesses the default gap size used when automatically placing elements within the containing visual element. The "autogap" defaults to zero.
autoplace[gap]	perform this.autoplace[42] perform this.autoplace	(Right, Number) Automatically places the template within the containing layout, subject to the standard rules for automatic placement. The gap parameter specifies the gap to be used between this template and the previous placed element. The parameter can be omitted, in which case the established "autogap" is utilized. Automatic placement for templates can only be used within layouts. A value of zero is always returned.

PORTAL CONTEXT

The "portal" context identifies a portal within the current hierarchy. Portals can either be used within templates or within layouts, so the parent context of the portal within the hierarchy can vary.

CONTEXT TRANSITIONS

From within a "portal" context, you can utilize the following set of valid context transitions:

Transition	Example	Definition
parent	this.parent	Transitions to the layout context or template context corresponding to the visual element that contains the portal. **NOTE!** The "parent" transition can only be utilized a single time, so it is not possible to go upwards two or more levels within the hierarchy.

TARGET REFERENCES

The "portal" script context governs the operations that can be applied to portals within templates and layouts. The complete list of target references for portals is presented in the table below.

Reference	Example	Definition
width	result = this.width this.width = 42	(Left, Right, Number) Accesses the width of the portal. Unless explicitly specified within the XML, the width of a portal is automatically initialized appropriately to the nature of the portal. Setting the width of a table portal triggers immediate re-sizing of all items within the table.
height	result = this.height this.height = 42	(Left, Right, Number) Accesses the height of the portal. Unless explicitly specified within the XML, the height of a portal is automatically initialized appropriately to the nature of the portal. Setting the height of a table portal triggers immediate re-sizing of all items within the table.
left	result = this.left this.left = 42	(Left, Right, Number) Accesses the position of the left edge of the portal within the containing visual element.
top	result = this.top this.top = 42	(Left, Right, Number) Accesses the position of the top edge of the portal within the containing visual element.
right	result = this.right	(Right, Number) Returns the position of the right edge of the portal within the containing visual element.
bottom	result = this.bottom	(Right, Number) Returns the position of the bottom edge of the portal within the containing visual element.
visible	result = this.visible this.visible = 1	(Left, Right, Number) Controls the visibility of the portal within the containing visual element. A non-zero value indicates the portal is visible and a zero value indicates hidden.
freeze	result = this.freeze this.freeze = 1	(Left, Right, Number) Controls whether the portal is displayed in a "frozen" state, where a frozen portal is non-editable and for display only. A non-zero value indicates the portal is frozen and a zero value indicates normal behavior. **NOTE!** All portals must be frozen before any portals are positioned.
scrollbar	result = this.scrollbar	(Right, Number) Returns the width of a scroller, in pixels.
sizetofit[min]	result = this.sizetofit[36]	(Right, Number) Reduces the font size, as necessary, to fit the text contents of the portal into the available space given by the portal dimensions. The font size will be reduced in quarter-point increments until either the text fits or the minimum font size given by min is reached. The value zero is always returned. **NOTE!** This target reference is only valid for use with the various types of label portals. **NOTE!** The minimum font size allowed is 6-point for screen output and 4-point for printed output.

Reference	Example	Definition
autotop	this.autotop = 42	(Left, Right, Number) Accesses the position of the top edge of the auto-place region within the containing visual element.
autobottom	result = this.autobottom this.autobottom = 420	(Left, Right, Number) Accesses the position of the bottom edge of the auto-place region within the containing visual element.
autoleft	result = this.autoleft this.autoleft = 42	(Left, Right, Number) Accesses the position of the left edge of the auto-place region within the containing visual element.
autoright	result = this.autoright this.autoright = 420	(Left, Right, Number) Accesses the position of the right edge of the auto-place region within the containing visual element.
autowidth	result = this.autowidth this.autowidth = 420	(Left, Right, Number) Accesses the width of the auto-place region within the containing visual element.
autoheight	result = this.autoheight this.autoheight = 420	(Left, Right, Number) Accesses the height of the auto-place region within the containing visual element.
autogap	result = this.autogap this.autogap = 42	(Left, Right, Number) Accesses the default gap size used when automatically placing elements within the containing visual element. The "autogap" defaults to zero.
autoplace[gap]	perform this.autoplace[42] perform this.autoplace	(Right, Number) Automatically places the portal within the containing layout, subject to the standard rules for automatic placement. The gap parameter specifies the gap to be used between this portal and the previously placed element. The parameter can be omitted, in which case the established "autogap" is utilized. Automatic placement for portals can only be used within layouts. A value of zero is always returned.

TABLE CONTEXT

The "table" context identifies the containing table of a template or portal within the current hierarchy.

CONTEXT TRANSITIONS

The "table" context is essentially the same as a "portal" context, since the context simply corresponds to a table portal within a layout. From within a "table" context, you can utilize the following set of valid context transitions:

Transition	Example	Definition
parent	this.parent	Transitions to the layout context corresponding to the visual element that contains the portal. **NOTE!** The "parent" transition can only be utilized a single time, so it is not possible to go upwards two or more levels within the hierarchy.

TARGET REFERENCES

Tables are a special type of portal. As such, they share all of the same target references as normal portals. However, tables also have a few additional target references that are unique to themselves. The complete list of these special target references is presented in the table below.

IMPORTANT! Tables also support all general portal target references.

Reference	Example	Definition
gapx	perform this.gapx	(Right, Number) Returns the gap between items within the table along the X-axis.

Reference	Example	Definition
gapy	perform this.gapy	(Right, Number) Returns the gap between items within the table along the Y-axis.

VALUE CONTEXT

The "value" context represents any field within the pick or thing associated with a template. The value context is used for display only, so all aspects of the field are always read-only.

CONTEXT TRANSITIONS

The "value" context parallels a "field" context, with the key distinction being that all values are read-only, since they are intended for display only. From within a "value" context, you can utilize the following set of valid context transitions:

Transition	Example	Definition
-None-	N/A	There are no transitions from within a value context.

TARGET REFERENCES

The "value" script context is actually the same as a "field" context, except that the field is being accessed from within a visual script and is therefore read-only in all behaviors. Hence the need to keep the script contexts distinct. The complete list of target references for the "value" context is presented in the table below.

Reference	Example	Definition
value	result = this.field[myfield].value	(Right, Number) Returns the contents of the field as a numeric value. If the field is text, the contents are automatically converted to a suitable value.
text	result = this.field[myfield].text	(Right, String) Returns the contents of the field as a string. If the field is numeric, the contents are automatically converted to a suitable string.
isempty	result = this.field[myfield].isempty	(Right, Number) Returns non-zero if the field contains the empty string and zero if it contains text of any length. If used on a numeric field or with an array or matrix, an error is reported. Testing of array and matrix elements must be done via the "compare" intrinsic.
ischanged	result = this.field[myfield].ischanged	(Right, Number) Returns non-zero if the field contents have been changed in any way from its original starting state. This can detect a field that has been changed by the user or via a script.
ischosen	result = this.field[myfield].ischosen	(Right, Number) Returns non-zero if the field contains a pick or a thing that was selected via a menu. If the field is associated with a menu and no selection has yet been made, zero is returned.
delta	result = this.field[myfield].delta	(Right, Number) Returns the delta component of a user field. The field must be explicitly configured to support delta processing.
arrayvalue[row]	result = this.field[myfield].arrayvalue[3]	(Right, Number) Returns the contents of a specific element of the array-based field as a numeric value. The index of the element is given by row, where the index is a zero-based value that must be less than the number of rows in the array. If the field is text, the contents are automatically converted to a suitable value.

Reference	Example	Definition
arraytext[row]	result = this.field[myfield].arraytext[3]	(Right, Number) Returns the contents of a specific element of the array-based field as a string. The index of the element is given by row, where the index is a zero-based value that must be less than the number of rows in the array. If the field is numeric, the contents are automatically converted to a suitable string.
matrixvalue[row,col]	result = this.field[myfield].matrixvalue[3,4]	(Right, Number) Returns the contents of a specific element of the matrix-based field as a numeric value. The index of the element is given by row and col, where the indices are zero-based values that must be less than the number of rows and columns in the matrix, respectively. If the field is text, the contents are automatically converted to a suitable value.
matrixtext[row,col]	result = this.field[myfield].matrixvalue[3,4]	(Right, String) Returns the contents of a specific element of the matrix-based field as a string. The index of the element is given by row and col, where the indices are zero-based values that must be less than the number of rows and columns in the matrix, respectively. If the field is numeric, the contents are automatically converted to a suitable string.
datetime[fmt,sep]	result = this.field[myfield].datetime[gamedate,"/"]	(Right, String) Returns the contents of the field formatted for output as a date or time, where sep is the separator string to use between values. The fmt parameter dictates the formatting to be used and must be one of the following values: • ***realdate:*** Formatted as if it's a real-world date. • ***realtime:*** Formatted as if it's a real-world time. • ***gamedate:*** Formatted as if it's a game-world date. • ***gametime:*** Formatted as if it's a game-world time. **NOTE!** This target reference is only supported on fields within picks - not things.
history[spl]		(Right, String) Generates and returns a text string that contains the change history for the field. The spl parameter is a string that is inserted between entries in the change history, splicing them together for appropriate output. The history report contents depend on the history behavior assigned to the field, as given below: • ***best:*** Only the notes text for each history entry is reported • ***stack:*** The notes text for each entry is reported and the adjustment details are included in parentheses with the notes (e.g. "notes (+2)") **NOTE!** This target reference is only supported on fields within picks - not things.

STATE CONTEXT

The "state" context provides access to overall state information that pertains to the portfolio as a whole or general information about the evaluation cycle.

CONTEXT TRANSITIONS

From within a "state" context, you can utilize the following set of valid context transitions:

Transition	Example	Definition
thing[id]	this.thing[myid]	Transitions to the thing context corresponding to the thing with the id specified. If no thing exists with the given unique id, an error is reported.

TARGET REFERENCES

The "state" context is a general context that maintains information outside the normal data hierarchy. The target references for this context span a wide range of details that may prove useful within your scripts. The complete list of target references for the "state" context is presented in the table below.

Reference	Example	Definition
isfocus	result = state.isfocus	(Right, Number) Returns non-zero if a focus pick has been properly established via the "setfocus" target reference.
clearfocus	perform state.clearfocus	(Right, Number) Clears any focus pick that has been established and resets to a state where there is no focus pick.
timing	result = state.timing	(Right, String) Returns the phase and priority timing during which the current script is being invoked in an effort to simplify debugging of data files.
iscreate	result = state.iscreate	(Right, Number) Returns non-zero if the character is currently in creation mode.
isadvance	result = state.isadvance	(Right, Number) Returns non-zero if the character is currently in advancement mode.
issell	result = state.issell	(Right, Number) Returns non-zero if the user is currently in the midst of a sell transaction.
isload	result = state.isload	(Right, Number) Returns non-zero if the loading of a saved portfolio is currently in progress.
isoutput	result = state.isoutput	(Right, Number) Returns non-zero if the rendering of character sheet output is currently in progress.
isdossierstyle[style]	result = state.isdossierstyle[html]	(Right, Number) Returns whether the user selected text output to be formatted in the style given by the style parameter. The style parameter must be one of the following values: "plain", "html", or "bbcode".
istext	result = state.istext	(Right, Number) Returns non-zero if the render of text output is currently in progress.
iscombat	result = state.iscombat	(Right, Number) Returns non-zero if combat mode is currently enabled within the Tactical Console.
isendturn	result = state.isendturn	(Right, Number) Returns non-zero if all combatants have taken their allotted actions and it is valid to end isendturn the current combat turn.
combatturn	result = state.combatturn	(Right, Number) Returns the current combat turn that is underway within the Tactical Console.

Reference	Example	Definition
initchange	result = state.initchange	(Right, Number) Returns non-zero if any actor has a user-modified initiative value.
actorcount	result = state.actorcount	(Right, Number) Returns the total number of actors within the portfolio, including all minions.
reload[table]	result = state.reload[myportal]	(Right, Number) Forces a re-load and re-sort of the table portal whose id is given by the table parameter. This target reference is only accessible from within a Trigger Script.
value[id]	result = state.value[id]	(Left, Right, Number) Provides direct access (read/write) to a global value with the given id. This value is globally defined, outside the scope of any actors, and it is persistent. The state of all global values is saved with the portfolio and restored on a reload. A global value must be set before it is retrieved, else a run-time error is reported. **NOTE!** This mechanism makes it possible to manage state across the entire portfolio.
setrandom[id,cnt]	result = state.setrandom[setid,42]	(Right, Number) Creates a new set of random values with the given id. The set contains the integer values zero through cnt-1, with the values being in a random sequence. The set of values is globally defined, outside the scope of any actors, and it is persistent. The state of the set will be saved with the portfolio and restored on a reload. The value returned is always zero. **NOTE!** This mechanism makes it possible to simulate set-based behaviors, such as a deck of cards. The other target references beginning with the "set" prefix below are used in conjunction with this set.
setextract[id]	result = state.setextract[setid]	(Right, Number) Extracts and returns the next value from the set with the given id, which must already be created. If there are no values left within the set, a run-time error is reported and the value zero is returned.
setremain[id]	result = state.setremain[setid]	(Right, Number) Returns the number of values that remain within the set with the given id.
setdiscard[id,val]	result = state.setdiscard[setid,42]	(Right, Number) Locates the value val within the set with the given id and discards it from the set. Once discarded, the set behaves as if the value no longer exists within the set.

CONTEXT TRANSITIONS

Every script begins with an initial context that is dictated by the particular script. Quite often, though, you'll want to access information somewhere else within the data hierarchy. That's when context transitions come into play. A context transition allows you to move through the hierarchy, progressing to objects either above or below the current context. These transitions can be chained, allowing you to move through a sequence of contexts to reach the desired destination.

NOTE! If transitions are utilized that result in an invalid (i.e. non-existent) context, any subsequent target reference will be invalid. If this occurs during run-time, the operation will be ignored and the target identifier will return zero. A suitable error will generally be displayed, but not always. An example of an invalid context is when a pick attempts to transition to a field that does not exist within that pick.

USING "THIS"

Every script has an initial context that is automatically established (see the specific script to know what it is). Normally, this context is implied, so you don't need to do anything to reference that context. However, some authors will want their scripts to clearly indicate when the implied context is being used. To accommodate this, scripts can utilize the reserved word "this" to indicate the implied context.

For example, an Eval Script starts with the pick as its implied context. So you could write a target identifier that checks the validity of that pick as simply "valid". Alternately, you could specify "this" as the context, yielding a target identifier of "this.valid". Either method is perfectly legal and you are welcome to use whichever method you prefer.

NOTE! The "this" reference identifies the implied context only. Therefore, you can only use "this" as the first context reference for a target identifier. If "this" is used anywhere else, a compilation error will occur.

TRANSITIONS BY CONTEXT

From within a given context, you are only able to transition to a specific set of other contexts. The topics below identify what the valid transitions are for each context.

- Container Context Transitions
- Hero Context Transitions
- Pick Context Transitions
- Thing Context Transitions
- Field Context Transitions
- Pool Context Transitions
- Scene Context Transitions
- Layout Context Transitions
- Template Context Transitions
- Portal Context Transitions
- Table Context Transitions
- Value Context Transitions
- State Context Transitions

SPECIAL CONTEXTS

Within some scripts, special contexts are supported. These special contexts behave the same way within any script that uses them. However, what they correspond to may be different within each script. The specific scripts where special contexts can be used will explicitly cite the availability of the context in their description. Alternately, some special contexts can be established within a script via certain language mechanisms (e.g. "eachpick"). The behavior of these contexts is outlined below:

- **eachpick:** The "eachpick" context only applies within the context of a "foreach" statement and represents the pick that is currently being iterated upon by the "foreach" statement. The current pick is accessed via the "eachpick." initial context. When used, the script context is switched to the iterated pick, and all subsequent actions are performed relative to that pick.
- **altpick:** The "altpick" context represents an alternate pick that is integral to the script and can be accessed readily. The alternate pick is accessed via the "altpick." initial context. When used, the script context is switched to the alternate pick, and all subsequent actions are performed relative to that pick.
- **altthing:** The "altthing" context represents an alternate thing that is integral to the script and can be accessed readily. The alternate thing is accessed via the "altthing." initial context. When used, the script context is switched to the alternate thing, and all subsequent actions are performed relative to that thing.
- **focus:** The "focus" context only applies when the "setfocus" target reference is utilized on a pick context to establish that pick as a memorized context. The established focus pick context is accessed via the "focus." initial context, after which the script context changes to that pick and all subsequent actions are performed relative to that pick.
- **actor:** The "actor" context only applies when the "setactor" target reference is utilized on a hero context to establish that hero as a memorized context. The established actor context is accessed via the "actor." initial context, after which the script context changes to that actor and all subsequent actions are performed relative to that actor. **IMPORTANT!** This is a situation where there is a critical distinction between "hero" and "actor". The "actor" context only applies to whatever actor has been established as the "actor focus".
- **transaction:** The "transaction" context represents an alternate pick that is integral to scripts involved in buy and sell transactions of objects. The alternate pick is accessed via the "transaction." initial context. When used, the script context is switched to the alternate pick, and all subsequent actions are performed relative to that pick.

IMPORTANT! All special contexts must be specified at the **start** of an identifier. If not, they will not be acknowledged by the compiler. For example, if the "altpick" special context is supported by a script, the reference "altpick.field[livename].text" would work perfectly. However, the reference "this.altpick.field[livename].text" would fail to compile, since "altpick." is not given as the initial context for the identifier.

TARGET REFERENCES

When managing data via scripts, the first step is to establish the correct script context. Once you've identified the desired context, you'll then need to specify the appropriate target reference to retrieve or manipulate the specific information you want. Every script context has an assortment of target references that provide access to data pertaining to that context.

NOTE! If the current script context is invalid, any target reference used within that context. If this occurs during run-time, the operation will be ignored and the target identifier will return zero. A suitable error will generally be displayed, but not always. An example of an invalid context is when a pick attempts to transition to a field that does not exist within that pick.

TARGET REFERENCE BEHAVIORS

Every target reference is assigned a set of behaviors that dictate how it can be used within scripts. The different behaviors are described in the table below:

- **Right:** This target reference can be utilized on the right side of an assignment statement or with the "perform" statement. The value returned is specified within its description.
- **Left:** This target reference can be utilized on the left side of an assignment statement. It must be assigned a value in accordance with its description.
- **Number:** This target reference utilizes a numeric value, either returning a number or expecting to be assigned a number.
- **String:** This target reference utilizes a text string, either returning a string or expecting to be assigned a string.

TOPICS

The topics below delineate the various target references that are available within each context.

- Container Target References
- Hero Target References
- Pick Target References
- Thing Target References
- Field Target References
- Pool Target References
- Scene Target References
- Layout Target References
- Template Target References
- Portal Target References
- Value Target References
- Table Target References
- State Target References

DATA ACCESS EXAMPLES

Let's look at a few examples of combining context transitions with target references into a useful target identifier. For the first example, we assume we're writing an Eval Script in which we need to access the calculated bonus conferred by the "strength" ability score for the hero. Since an Eval Script starts out in the context of a pick, we'll just go directly to the hero and drill down to get the value we need. From the hero, we transition to the pick that contains the "strength" ability score. Once we have the pick, we then transition to the "bonus" field. And finally we access the value of that field. In the end, the complete reference might look like the following:

```
hero.child[strength].field[bonus].value
```

What if our hero has a magic sword from which we need to extract information? In this situation, we need to drill down into the child gizmo and then into a pick belonging to that gizmo. Starting from the root hero again, we first access the pick that attaches the gizmo. From there, we transition to the gizmo, after which we can transition to the pick inside the gizmo. Once we have the pick context, we can get the field and access its value. The final result would look something like the following:

```
hero.child[magicsword].gizmo.child[attack].field[bonus].value
```

Now let's assume we have a script on a pick within the above gizmo that needs to access a field on a different pick within the same gizmo. Theoretically, we could bounce all the way out to the hero and drill down, but that approach won't work reliably if we have multiple magic swords on the hero. So the best approach is to move upwards to the container and then back down again into the desired pick. The process would look something like the following:

```
container.child[attack].field[bonus].value
```

The key thing to remember when writing scripts is to always know your initial context. From there, simply identify where you want to reference and then systematically transition, one step at a time. If you take things one step at a time, you'll be able to access the information you need and your scripts will work smoothly.

SCRIPT TYPES

The Kit leverages a diverse assortment of scripts for a wide range of purposes. The topics below provide a brief discussion of both the role and behavior of each different type of script. The scripts have been grouped into general categories for improved utility.

PICK MANIPULATION

These scripts manipulate the contents of picks during the evaluation cycle.

EVAL SCRIPT

TECHNICAL DETAILS

Initial Context:			Pick
Alternate Context:			None
Fields Finalized?			No
Where Used:		Components,	Things
Procedure Use: "eval" type, "pick" context			

The Eval script utilizes the following special symbols:

-None- There are no special symbols for an Eval script.

DESCRIPTION

The Eval script is the primary workhorse throughout any set of data files. This script is scheduled within the evaluation cycle and where you will orchestrate the virtually all of the effects for the game system. If a special ability confers a bonus to certain skills, you'll use an Eval script to apply them. If attributes confer adjustments to abilities, attacks, damage, or anything else, you'll use Eval scripts to apply them. Calculated abilities, such as attack ratings or casting powers will be determined through Eval scripts. The list is endless.

Every Eval script is associated with a particular thing. If an Eval script is defined for component, then it is inherited by the things which derive from that component. When a thing is added to a container, it becomes a pick, and new instances of every Eval script for that thing are created and managed by HL. You associate all of the primary behaviors with each pick via the Eval scripts defined for the underlying thing.

When invoked, an Eval script starts with its associated pick as its initial context. You are free to navigate through the hierarchy and effect changes anywhere you deem appropriate. However, all of the changes will typically either be made to the associated pick or be made to other objects based on facets of the associated pick.

As a general rule, all of the dynamic effects that are applied throughout your data files should be handled via Eval scripts. Because the timing of all Eval scripts is controlled, you can ensure that all of the different effects of different scripts are applied in a carefully defined sequence. For example, in the d20 System, adjustments to attributes (e.g. from magic items) must be applied before the bonuses conferred by attributes are calculated and applied.

Eval scripts provide a single, generalized mechanism through which you can accomplish just about anything. As a result, the actual behaviors of an Eval script will vary more widely than with any other type of script. If there is one type of script to get comfortable with first, it is definitely the Eval script.

EXAMPLE

Due to the wide range of behaviors that can be applied via Eval scripts, there is no one good example. The XML below shows three separate Eval scripts that are defined as part of the basic handling of equipment within the Sample data files. Each script is scheduled to be performed at a different time during the overall evaluation cycle, with the timing ensuring that each behavior is properly interleaved with other behaviors triggered by other objects.

```
<!-- All melee weapons get the appropriate tag -->
 <eval index="1" phase="Setup" priority="5000"><![CDATA[
   perform assign[Armory.Melee]
   ]]></eval>

<!-- Calculate the net attack roll for the weapon -->
 <eval index="2" phase="Final" priority="7000" name="Calc wpNetAtk"><![CDATA[
   field[wpNetAtk].value = #trait[skMelee] + field[wpBonus].value + field[wpPenalty].value
```

```
 ]]></eval>


<!-- Prepend any derived special notes to the appropriate field -->
 <eval index="3" phase="Render" priority="2000"><![CDATA[
  var special as string


  ~assign any appropriate special notes to the "special" variable here


  ~prepend any existing special details with the notes for this weapon
  if (empty(special) = 0) then
    if (field[wpNotes].isempty = 0) then
      special &= ", "
      endif
    field[wpNotes].text = special & field[wpNotes].text
    endif
  ]]></eval>
```

GEAR SCRIPT

TECHNICAL DETAILS

Initial	**Context:**	Pick
Alternate	**Context:**	None
Fields	**Finalized?**	No
Where	**Used:**	Things

Procedure Use: None

The Eval script utilizes the following special symbols:

-None- There are no special symbols for a Gear script.

DESCRIPTION

The purpose of the Gear script is to let you apply non-standard behaviors to gear. By default, all gear that is placed within a holder has its weight accrued into the net weight of the holder. For example, if a backpack normally weighs 3 pounds and there is a potion inside the backpack that weighs 2 pounds, the net weight of the backpack will be 5 pounds. However, there are some items that break this rule. The classic example is the Bag of Holding from the d20 System, which weighs a fixed amount, regardless of what is stored within it. The Gear script allows you to override the standard behaviors and assign a custom weight to the holder.

The Gear script begins with an initial context of the pick itself. In the example above, the Gear script would start with the Bag of Holding as its initial context. From there, you can access other facets of the structural hierarchy if you need them. Upon entry to the script, the "gearHeld" field value has been automatically calculated to be the total weight of all items held within. In addition, the "gearNet" field value contains the net weight of the item, including the weight of both the item itself and its contents. By setting the values of these fields appropriately in the script, the appropriate weights will propagate up the containment hierarchy.

EXAMPLE

We'll continue with the example of the Bag of Holding here. For this item, you would simply set the weight of the pick to be its basic weight, ignoring the accrued weight of its contents.

```
~Our contents count for no weight at all!
 field[gearNet].value = field[gearWeight].value
```

FIELD MANIPULATION

These scripts manipulate the contents of fields for both display and constraint.

BOUND SCRIPT

TECHNICAL DETAILS

Initial	**Context:**	Pick
Alternate	**Context:**	None
Fields	**Finalized?**	No
Where	**Used:**	Field

Procedure Use: "bounds" type, "pick" context

The Bound script utilizes the following special symbols:

- **minimum:** (Number) Entry: The default minimum value to be used for bounding. Exit: The final minimum value to be used for bounding.
- **maximum:** (Number) Entry: The default maximum value to be used for bounding. Exit: The final maximum value to be used for bounding.

DESCRIPTION

The Bound script allows you to dynamically calculate the appropriate minimum and maximum values that a field can possess. This is invaluable when the minimums are dependent upon dynamically changing criteria. For example, consider the height and weight of a character. If the game system supports different races, then each race will have its own minimums and maximums for these characteristics. You need to be able to set the bounds based on the race that has been selected.

The Bound script begins with an initial context of the pick containing the field being bounded. This makes it possible to easily access other fields on the pick to base the bounding limits. The most common use of this script is for the containing pick to have separate fields that dictate the minimum and maximum values to be imposed. These fields are properly setup by Eval scripts and then the Bound script simply assigns those values as the limits.

When the Bound script is invoked, the minimum and maximum limits start out as the limits specified via the "minimum" and "maximum" attributes on the field definition. If no limits are defined via those attributes, then there is no limit on the field value. This makes it possible to have your Bound script use the defaults and only impose appropriate limits when necessary, leaving the limits unchanged otherwise.

Until a Bound script is invoked, the value of the field is bounded against the default limits specified by the field attributes. So any use of the field value within a script that occurs earlier than the Bound script will be restricted based on those limits. Once the Bound script is invoked, the current field value is immediately bounding to the new limits. Thereafter, any changes applied to the field value are automatically bounded to the new limits.

EXAMPLE

The bounding of age, height, and weight uses the approach outlined above, where separate fields are used to dictate the appropriate bounding limits. As such, the Bound script for the character's age simply pulls its limits from the separate fields, as shown below.

```
@minimum = field[perAgeMin].value
@maximum = field[perAgeMax].value
```

CALCULATE SCRIPT

TECHNICAL DETAILS

Initial	**Context:**	Pick
Alternate	**Context:**	None
Fields	**Finalized?**	No
Where	**Used:**	Fields

Procedure Use: "calculate" type, "pick" context

The Bound script utilizes the following special symbols:

- **value:** (Number) Entry: The current value of the field (if the field is numeric), else zero. Exit: The final value to be used for the field (if the field is numeric).
- **text:** (String) Entry: The current context of the field (if the field is text), else the empty string. Exit: The final contents to be used for the field (if the field is text).

DESCRIPTION

The Calculate script is in many ways a specialized Eval script, with the explicit purpose of calculating the value of a field. The script is scheduled to occur at some point during the evaluation cycle, and it could just as easily be implemented as an Eval script. The advantage of the Calculate script is purely a semantic organizational benefit, since the script is defined directly on the field instead of generally on the component (as an Eval script would be handled).

The Calculate script allows you to calculate the appropriate contents for a field. Since the script is scheduled during evaluation, other scripts can legally modify the field before or after the script. Consequently, it is usually only appropriate to use a Calculate script for fields that can be calculated in one centralized location (this script). Otherwise, it is typically better to use Eval scripts for all manipulations of the field.

The Calculate script begins with an initial context of the pick containing the field being calculated. This makes it possible to easily access other fields on the pick when calculating the new contents. When invoked, the appropriate special symbol is initialized with the current contents of the field. If the field is numeric, the "value" special symbol contains the value, and the "text" special symbol contains the

current contents of a text field. When the script completes, the updated contents of the appropriate special symbol are used for the field, with the other special symbol completely ignored.

EXAMPLE

Within the Sample data files, the final value of a trait is determined via a Calculate script. The field value is the result of adding the user-assigned value, any bonuses assigned via scripts, and any in-play adjustment.

```
@value = field[trtUser].value + field[trtBonus].value + field[trtInPlay].value
```

FINALIZE SCRIPT

TECHNICAL DETAILS

Initial	**Context:**	Pick
Alternate	**Context:**	None
Fields	**Finalized?**	No
Where	**Used:**	Fields

Procedure Use: "finalize" type, "pick" context

The Finalize script utilizes the following special symbols:

- **value:** (Number) Entry: The current value of the field (if the field is numeric), else zero. Exit: Ignored.
- **text:** (String) Entry: The current contents of the field (if the field is text), else the field value converted to a string. Exit: The contents to be used for finalized version of the field. The text may contain encoding.
- **ispick:** (Number) Entry: Indicates whether the value is being finalized for a pick (non-zero) or a thing (zero). Exit: Ignored.

DESCRIPTION

The role of the Finalize script is to synthesize a text version of a numeric field that consists of more than just the field value. By default, a numeric field is automatically converted to a string when the ".text" target reference is used on the field, so this script is of no use unless you want a different behavior.

There are numerous places where a Finalize script can be extremely handy. Lots of fields are managed as numeric values for easy of manipulation in data files. However, many of those fields need to be displayed to the user with appropriate units annotated. For example, the cash possessed by a character and the costs of equipment are tracked as numeric values but should ideally be displayed as currency (e.g. "$142"). The height of a character may be tracked as inches, but the user wants to view it in a more common form (e.g. 5'11"). The Finalize script makes it easy to accomplish this.

The Finalize script is invoked after the entire evaluation cycle has completed, but it is only invoked when something actually asks for the finalized contents of the field. Once the script is invoked once, the results are cached internally, but the cost of invoking the script is not incurred unless the contents are actually used somewhere.

The Finalize script begins with an initial context of the pick containing the field being calculated. When invoked, the "value" special symbol is assigned the actual value of the field at the end of evaluation. If the field is text, then the value is always zero. Similarly, the "text" special symbol is assigned the text-based contents of the field after evaluation. If the field is numeric, the value is automatically converted to the corresponding string. When the script completes, the updated contents of the "text" special symbol are used as the finalized contents for the field.

NOTE! When accessing a numeric field via a script, you have the option to use either the ".value" or ".text" target reference. Within any script that utilizes finalized field values (indicated at the top of each script), you should always use the ".text" target reference, unless you have a specific reason for using the ".value" target reference. The reason for this is that ".text" will always retrieved any finalized version of the field, while ".value" will always retrieve the actual value, without any finalization logic being applied.

EXAMPLE

Within the Sample data files, the height of a character is managed as a numeric field that handles everything in terms of inches. The user wants to view this in the standard format using feet and inches (e.g. 5'11"). A Finalize script is defined that accomplishes this, as shown below.

```
~calculate the height in terms of feet and inches
  var feet as number
  var inches as number
  feet = @value / 12
  feet = round(feet,0,-1)
  inches = @value - (feet * 12)

  ~synthesize appropriate text to display the height properly
```

```
@text = feet & """
if (inches <> 0) then
  @text = @text & " " & inches & chr(34)
  endif
```

INITFINALIZE SCRIPT

TECHNICAL DETAILS

Initial		**Context:**	Pick
Alternate		**Context:**	None
Fields		**Finalized?**	No
Where	**Used:**	Global	Behavior

Procedure Use: "finalize" type, "pick" context

The InitFinalize script utilizes the following special symbols:

- **value:** (Number) Entry: The current value of the initiative determined for the actor. Exit: Ignored.
- **text:** (String) Entry: The current initiative value converted to a string. Exit: The contents to be used for finalized version of the field. The text may contain encoding.

DESCRIPTION

The InitFinalize script is a special-purpose instance of a normal Finalize script. The value being finalized is the initiative value calculated for an actor via the Initiative script. In all respects, this script behaves as a standard Finalize script, except that there are no other field values that can be accessed via the initial pick context.

EXAMPLE

This is a standard Finalize script, so the example for that script can be referenced

VALIDATION

These scripts apply validation tests to objects with integrated reporting of errors.

EVALRULE SCRIPT

TECHNICAL DETAILS

Initial		**Context:**	Pick
Alternate		**Context:**	None
Fields	**Finalized?**		No
Where	**Used:**	Components,	Things

Procedure Use: "evalrule" type, "pick" context

The EvalRule script utilizes the following special symbols:

- **valid:** (Number) Entry: Assumed the rule is not satisfied with a value of zero. Exit: Indicates whether the rule is satisfied (non-zero) or not satisfied (zero).
- **message:** (String) Entry: Contains the default message text if the rule is not satisfied. Exit: Contains the final message text to display if the rule is not satisfied. The text may contain encoding.
- **summary:** (String) Entry: Contains the default summary text if the rule is not satisfied. Exit: Contains the final summary text to display if the rule is not satisfied. If not specified, but the "message" symbol is modified, the summary automatically uses the message text.

DESCRIPTION

The EvalRule script is very similar to an Eval script, with the primary distinction being that the EvalRule script verifies a rule is observed instead of applying changes. This script is scheduled during the evaluation cycle, so it allows you to check the state of an actor at intermediate points, in addition to at the end of evaluation. Intermediate checks make it possible to verify values before they are further modified by other aspects of the character.

An EvalRule script shares all the same behaviors of an Eval script, plus it adds the rule to be validated. The rule can be anything you want. However, the rule is always either satisfied or not satisfied. This is determined by the "valid" special symbol. The symbol starts out as zero, indicating that the rule is not satisfied. At any point during the script, you can set the symbol to non-zero, thereby indicating that the rule is satisfied. You can also change the symbol value throughout the script if you want. The only value that matters is the final value when the script ends, where a non-zero value indicates the rule is satisfied.

If the validation test is not satisfied, the message and summary for the rule are displayed within the validation report and validation summary for the actor, respectively. Since the message and summary are typically static requirements (e.g. an attribute must be a value of X or more), you will rarely need to modify them. However, there will be cases where you want to tailor the message and/or summary to provide more detailed information based on dynamic information. When these situations arise, simply set the "message" and/or "summary" special values to the desired contents and they will be used.

Since an EvalRule script is equivalent to an Eval script, it is possible to apply changes throughout the structural hierarchy via the script. However, it is generally a bad idea to do that, since the purpose of an EvalRule script is to validate a rule - not effect change. We therefore recommend that you avoid applying changes within EvalRule scripts.

Like an Eval script, every EvalRule script is associated with a particular thing. If an EvalRule script is defined for a component, then it is inherited by the things which derive from that component. When a thing is added to a container, it becomes a pick, and new instances of every EvalRule script for that thing are created and managed by HL. You associate all of the validation rules with each pick via the EvalRule scripts defined for the underlying thing.

When invoked, an Eval script starts with its associated pick as its initial context. You are free to navigate through the hierarchy and test the state of anything that impacts the rule. However, a given EvalRule script should typically either pertain to the associated pick or pertain to the actor as a whole.

EXAMPLE

An EvalRule script can be used to test virtually anything. Within the Sample data files, a rule is used to ensure that pieces of gear are not both equipped and held within a container (e.g. a sword being equipped and stored in a backpack). The rule below will verify this condition.

```
~if not both equipped and held within a container, we're valid
if (field[grIsEquip].value + isgearheld < 2) then
  @valid = 1
   Done
   endif

~mark the tab as invalid
 linkvalid = 0
```

VALIDATE SCRIPT

TECHNICAL DETAILS

Initial Context:				Container
Alternate Context:		Pick	or	Thing
Fields Finalized?				Yes
Where Used:		Components,		Things

Procedure Use: "validate" type, "pick" context

The Validate script utilizes the following special symbols:

- **valid:** (Number) Entry: Assumes the pre-requisite is not satisfied with a value of zero. Exit: Indicates whether the pre-requisite is satisfied (non-zero) or not satisfied (zero).
- **message:** (String) Entry: Contains the default message text if the pre-requisite is not satisfied. Exit: Contains the final message text to display if the pre-requisite is not satisfied. The text may contain encoding.
- **ispick:** (Number) Entry: Indicates whether the pre-requisite is being applied to a pick (non-zero) or thing (zero). Exit: Ignored.

DESCRIPTION

The Validate script is exclusively used within pre-requisite tests. The script verifies that a specific pick or thing satisfies a particular pre-requisite relative to its container. If the requirement is not satisfied, then the object is designated as invalid. If the object is a pick that has already been added to the portfolio, the specified message is displayed within the validation report for the actor.

The pre-requisite test can be anything you want, provided only the object and the container are considered by the test. The requirement is always either satisfied or not satisfied. This is determined by the "valid" special symbol. The symbol starts out as zero, indicating that the requirement is not satisfied. At any point during the script, you can set the symbol to non-zero, thereby indicating that the requirement is satisfied. The only value that matters is the final value when the script ends, where a non-zero value indicates the pre-requisite is satisfied.

Each pre-requisite is associated with a particular thing. If a pre-requisite is defined for a component, then it is inherited by the things which derive from that component. The pre-requisites are checked for every thing when it is presented as an option for the user to add via a table or chooser. If one or more pre-requisites are not satisfied, the thing is designated as invalid and the failed requirements are shown in the description information for the thing. If the thing is added by the user in spite of the failed pre-requisites, then all the pre-requisites are applied to the pick at the end of every evaluation cycle.

When invoked, a Validate script starts with the container as its initial context. If the object is a pick, the container is the one the pick resides within. If the object is a thing, the container is the prospective container to which the thing will potentially be added. The reason for the container as the initial context is that the pre-requisite test is defined on the thing. Consequently, the thing will know about itself and will typically be looking to verify that the prospective container satisfies the needs of the thing.

For those situations where you also need to access characteristics of the pick or thing, the Validate script provides an alternate context. This context will always be the pick or thing, as appropriate. You can use the "ispick" special symbol to determine whether the pre-requisite is being applied to a pick or a thing. After that, you can use the "altpick" or "altthing" context to access the object.

NOTE! The use of pre-requisites with a Validate script is generally only needed in more complex situations. The "pickreq" and "exprreq" mechanisms are much simpler to use and maintain, and they should cover 90% of the situations where you need to establish a pre-requisite.

EXAMPLE

A simple pre-requisite might establish a dependency on an attribute value being at least equal to some number. The example below shows a Validate script that tests whether the container has a child pick (strength) that has a final value of at least 13.

```
if (child[attrStr].field[trtFinal].value >= 13) then
  @valid = 1
  endif
```

INTEGRITY SCRIPT

TECHNICAL DETAILS

Initial Context:		Container
Alternate Context:		None
Fields Finalized?		Yes
Where Used:		Entities
Procedure Use: "container" context		

The Integrity script utilizes the following special symbols:

- **message:** (String) Entry: Contains the empty string to indicate that no errors were identified. Exit: Contains the final error message text to display. If there are no errors, specify the empty string. The text may contain encoding.

DESCRIPTION

The Integrity script is associated with entities. This script allows the author to verify that all the appropriate values have been specified for a gizmo before allowing the user to actually save changes to the gizmo.

The Integrity script is invoked when the user finishes editing a gizmo derived from the entity for which the script is defined. When the user tries to save his changes, the script is triggered, at which point the script verifies that an assortment of characteristics are satisfied by the gizmo. If any are not satisfied, an error message is displayed to the user and the user is not allowed to save his changes until the errors are corrected.

It is the responsibility of the script to synthesize the message shown to the user that reports the errors in the gizmo. If an error is detected in the script, a suitable error message should be appended to the "message" special symbol. If multiple errors occur, the message should contain a list of them, with each message on new line. If there are no errors, then the special symbol should be the empty string, and that tells HL that the gizmo is safe to save.

When invoked, an Integrity script starts with the gizmo as its initial context. There is generally no reason to navigate outside the context of the gizmo, but it is technically allowed. The primary focus will be on the gizmo itself and the child picks within it.

NOTE! Don't be too stringent with the Integrity script. Remember that every gaming group has its own house rules that will modify the basic game rules. As such, it's important to use validation rules to trap most errors instead of requiring the user to obey certain rules. The Integrity script should only be used to enforce details that should always be required and/or that impact your ability to write quality data files.

EXAMPLE

The handling of advancement that is found within the Sample data files utilizes an Integrity script. Based on the information shown for a particular advancement, certain fields must be specified by the user. If that information is not provided, the form remains shown and no changes to the gizmo are saved.

```
~setup a bullet character we can put at the start of each error
 var bullet as string
```

```
bullet = "{bmp bullet_red}{horz 4}"

~verify that a chooser selection is made if one is required
if (parent.tagis[Advance.MustChoose] <> 0) then
  if (firstchild["Advance.Gizmo"].tagis[Advance.Gizmo] = 0) then
    @message = @message & bullet & "A specific trait/ability must be selected via the chooser.\n"
    endif
  endif

~verify that we've been given a domain if one is required
if (parent.tagis[Advance.MustChoose] + parent.tagis[Advance.AddNew] >= 2) then
  if (firstchild["Advance.Gizmo"].tagis[User.NeedDomain] <> 0) then
    if (empty(child[advDetails].field[advUser].text) <> 0) then
      @message = @message & bullet & "The selection requires that you specify an appropriate domain.\n"
      endif
    endif
  endif
```

CREATION/DELETION

These scripts perform appropriate setup and cleanup of specialized objects.

CREATION SCRIPT

TECHNICAL DETAILS

Initial	**Context:**	Pick
Alternate	**Context:**	None
Fields	**Finalized?**	No
Where	**Used:**	Components

Procedure Use: None

The Integrity script utilizes the following special symbols:

- None: There are no special symbols for a Creation script.

DESCRIPTION

The Creation script is defined for components, and it pertains to all picks that derived from a given component. The script is invoked a single time when a pick is first created. As such, its primary use is for performing special setup and/or configuration of picks.

For example, in the d20 System, there is an optional rule where characters always receive maximum hit points for each new level. This can be handled quite easily via a Creation script. If the user has enabled the setting, the hit points are set to the maximum, else they are set to zero so that the user needs to modify them appropriately.

When invoked, a Creation script starts with the pick as its initial context. There is generally no reason to navigate outside the context of the pick, but it is technically allowed. The primary focus will be on the pick and its fields.

EXAMPLE

Using the d20 System example cited above, the Creation script below will properly initialize the hit points field for a new class level based on the configuration setting.

```
if (hero.tagis[source.MaxHP] <> 0) then
  field[lvlHP].value = field[lvlHitDice].value
else
  field[lvlHP].value = 0
  endif
```

DELETION SCRIPT

TECHNICAL DETAILS

Initial	**Context:**	Pick
Alternate	**Context:**	None
Fields	**Finalized?**	No
Where	**Used:**	Components

Procedure Use: None

The Integrity script utilizes the following special symbols:

- **None:** There are no special symbols for a Deletion script.

DESCRIPTION

The Deletion script is the analog of the Creation script. The script pertains to all picks that are derived from a given component, and it is invoked a single time just before a pick is finally destroyed. As such, its primary use is for performing special wrapup handling of picks.

This script is rarely used, but it is quite necessary in those places. The easiest example is with usage pools, such as those used by the journal mechanism. Each journal pick adds its awards (e.g. XP, cash) to the overall usage pools maintained for the actor. But if the user deletes a journal pick, it's critical that the awards conferred by the pick be subtracted back out. Enter the Deletion script, which serves perfectly for this purpose.

When invoked, a Deletion script starts with the pick as its initial context. There is generally no reason to navigate outside the context of the pick, but it is technically allowed. The primary focus will be on the pick and its fields.

EXAMPLE

Using the example cited above, the Deletion script below will properly subtract out the adjustments from a journal pick before it is destroyed.

```
perform hero.usagepool[TotalXP].adjust[-usagepool[JrnlXP].value]
perform hero.usagepool[TotalCash].adjust[-usagepool[JrnlCash].value]
```

VISUAL POSITIONING

These scripts manage the size and positioning of visual elements within panels and sheets.

POSITION SCRIPT

TECHNICAL DETAILS

Initial Context:	Scene	or	Layout	or	Template
Alternate Context:					None
Fields Finalized?					Yes
Where Used:	Templates,	Layouts,	Panels,	Forms,	Sheets

Procedure Use: None

The Position script utilizes the following special symbols:

- **None:** There are no special symbols for a Position script

DESCRIPTION

The location of visual elements on the display is controlled via the Position script. This script is used the same way within scenes, layouts, and templates. Through this script, the visual elements contained within are sized and positioned. For example, a Position script for a template will coordinate the sizing and positioning of the various portals that are defined within the template.

Most visual elements are sized and positioned by their containing element. For example, a layout is often told by the containing scene how much space it gets to utilize and then it sizes it contents to fit within that space. However, there are times when visual elements must size themselves, so it is also possible for a visual element to size itself within its Position script. An example of this is where you have a template within a table that needs to determine its own height based on the information being shown within.

Within a visual container, the position of elements is always relative to the upper left corner of the container. If the container has a margin assigned, then upper left corner is adjusted accordingly. This ensures that the visual container can be positioned anywhere within its container, and the visual elements within don't need any knowledge of the container's position.

IMPORTANT! The Position script for a template within a table is invoked separately for each item shown within the table. If the template is within a table containing items of non-varying height (the norm), the Position script is also invoked one additional time. The occurs before each item is processed and serves the purposes of calculating the of each item. When invoked for sizing, all text- based fields are empty and all value-based fields have a value of zero. The "issizing" target reference allows detection of this special use.

NOTE! The Position script is read-only with respect to the contents of the portfolio, although visual elements may be modified. Within this script, all aspects of the hierarchy can be accessed, but nothing can be changed.

EXAMPLE

Assume we have a template that contains two portals: a label portal and an edit portal. The template is used to let the user edit the name of something, so the label portal displays "Name:" and the edit portal allows the user to edit the name. The label portal will be automatically sized to the width of its text, so all we need to do is size the edit portal and position both portals. The Position script for a simple template like this might look like the following.

```
~center both portals vertically
 perform portal[label].centervert
 perform portal[edit].centervert

~pick a width for the edit portal
 portal[edit].width = 150

~put the label on the left and the edit portal on its right
 portal[label].left = 0
 perform portal[edit].alignrel[ltor,label,10]
```

HEADER SCRIPT

TECHNICAL DETAILS

Initial		**Context:**	Template
Alternate		**Context:**	None
Fields		**Finalized?**	Yes
Where		**Used:**	Templates
Procedure Use: None			

The Header script utilizes the following special symbols:

- **None:** There are no special symbols for a Header script

DESCRIPTION

The Header script behaves very similarly to the Position script. This script's purpose is to handle the sizing and positioning of visual elements in one specific situation. When the same template is used within a table for positioning both items and a header, the Header script serves to position the header portals. If two separate templates are used for items versus header, the Position script for each is used and the Header script is not employed.

Although the behaviors are similar, the Header script is distinct from the Position script in a variety of ways. First of all, portals within the template that are designated as "isheader" are only accessible via the Header script, hence the name. Second, the Header script is invoked after the normal Position script. This allows the header contents to be sized and positioned relative to the final locations for non-header portals. To make this easy, the Header script can access both header and non-header portals, although any positioning changes to non-header portals is ignored.

For more details, please see the section on dual-purpose headers.

EXAMPLE

We'll assume that there are two fields within each item of the table that we want to put a simple header above. Let's call them "damage" and "range", and each is shown within its own portal. These two portals are positioned via the Position script, so the Header script will position the header labels directly above those portals.

```
~our header height is the height of our labels
 height = portal[hdrdamage].height

~if this is a "sizing" calculation, we're done
 if (issizing <> 0) then
   done
   endif

~each of our header labels has the same width as the corresponding data beneath
 portal[hdrdamage].width = portal[damage].width
 portal[hdrrange].width = portal[range].width

~center each header label on the corresponding data beneath
 perform portal[hdrdamage].centeron[horz,damage]
 perform portal[hdrrange].centeron[horz,range]

~align all header labels at the bottom of the header region
 perform portal[hdrdamage].alignedge[bottom,0]
 perform portal[hdrrange].alignedge[bottom,0]
```

SYNTHESIS & PRESENTATION

These scripts synthesize information for display to the user in some fashion, including labels, descriptions, mouse-over information, and stat blocks.

DESCRIPTION SCRIPT

TECHNICAL DETAILS

Initial Context:	Pick	or	Thing
Alternate Context:			None
Fields Finalized?			Yes
Where Used:			Portals

Procedure Use: "description" type, "info" context, "pick" context

The Description script utilizes the following special symbols:

- **text:** (String) Entry: Contains the default description for the object, which consists of the basic description text assigned to the object and any failed pre-requisites. Exit: Contains the text to be displayed to the user as the description. The final text may contain encoding.
- **ispick:** (Number) Entry: Indicates whether the text is being rendered for a thing (zero) or a pick (non-zero), allowing different handling to be performed for the two separate cases. Exit: Ignored.

DESCRIPTION

Whenever the user is presented with a selection list to choose from, the list of available items is shown on the left and a description area is provided on the right. The description area contains all the details for the currently highlighted item. By default, this consists of the basic description text assigned to the item and any pre-requisites failed by the item. However, the author can present more detailed information via the use of a Description script. This allows the author to factor in context-driven material, such as pertinent field values and adjusted values due to the influence of previous selections.

NOTE! The Description script is read-only. Within this script, virtually all aspects of the structural hierarchy can be accessed, but nothing can be changed.

EXAMPLE

When displaying weapons, it's useful to show the damage and range. This can be easily appended to the default text for display.

```
@text &= "\n"
@text &= "Damage: " & field[wpDamage}.text & "\n"
@text &= "Range: " & field[wpRange].text & "\n"
```

MOUSEINFO SCRIPT

TECHNICAL DETAILS

Initial Context:	Pick	or	Thing
Alternate Context:			None
Fields Finalized?			Yes
Where Used:			Portals

Procedure Use: "mouseinfo" type, "info" context, "pick" context

The MouseInfo script utilizes the following special symbols:

- **text:** (String) Entry: Contains the default mouse-info for the object, which consists of the basic description text assigned to the object and any failed pre-requisites. Exit: Contains the text to be displayed to the user as the mouse-info text. The final text may contain encoding.
- **Ispick:** (Number) Entry: Indicates whether the text is being rendered for a thing (zero) or a pick (non-zero), allowing different handling to be performed for the two separate cases. Exit: Ignored.

DESCRIPTION

HL makes extensive use of visual elements that the user can move the mouse over to obtain further details about an object. This is generally referred to as mouse-over highlighting, or simply mouse info, and it is accomplished via the MouseInfo script on various portals. The purpose of the MouseInfo script is to synthesize the actual text to be displayed for the pick/thing that the user is inquiring about.

NOTE! The MouseInfo script is read-only. Within this script, virtually all aspects of the structural hierarchy can be accessed, but nothing can be changed.

EXAMPLE

When displaying weapons, it's useful to show the damage and range. This can be easily appended to the default text for display.

```
@text &= "\n"
@text &= "Damage: " & field[wpDamage}.text & "\n"
@text &= "Range: " & field[wpRange].text & "\n"
```

LABEL SCRIPT

TECHNICAL DETAILS

Initial	**Context:**	Pick	or	Thing
Alternate		**Context:**		None
Fields		**Finalized?**		Yes
Where		**Used:**		Portals

Procedure Use: "label" type, "info" context, "pick" context

The Label script utilizes the following special symbols:

- **text:** (String) Entry: Contains the pre-defined text for the label portal. Exit: Contains the text to be displayed to the user as the label portal contents. The final text may contain encoding.
- **ispick:** (Number) Entry: Indicates whether the text is being rendered for a thing (zero) or a pick (non-zero), allowing different handling to be performed for the two separate cases. Exit: Ignored.

DESCRIPTION

The Label script is used exclusively within label portals, and it is the script defined for just-in-time rendering of the portal contents. Instead of pulling the information for display out of a field or using literal text, a script is defined. This script is invoked every time the display updates, allowing the script to re-calculate the information to be displayed based on the most recent state of the portfolio.

NOTE! The Label script is read-only. Within this script, virtually all aspects of the structural hierarchy can be accessed, but nothing can be changed.

IMPORTANT! Avoid using Label scripts indiscriminately. Since the script must be re-evaluated and re-displayed with every update of the interface, they can be computationally expensive and slow down performance on slower computers. In many cases, a separate field can be used in which the appropriate value can be calculated and placed. When a field is used, HL will only update the display if the value actually changes.

EXAMPLE

Within the d20 System data files, the hero's HP and AC are both shown at the top. Separate label portals could be used for each, but the width of the HP will vary from one to three digits, requiring scripts to re-calculate the positions of the portals all the time in case the number of digits has changed. It can be more efficient to simply use a single Label script that synthesizes the text to be displayed and eliminates all the re-positioning, resulting in a script like the one below.

```
@text = "HP: " & herofield[acHP].text & " AC: " & herofield[acArmorCls].text
```

TITLEBAR SCRIPT

TECHNICAL DETAILS

Initial	**Context:**	Container
Alternate	**Context:**	None
Fields	**Finalized?**	Yes
Where	**Used:**	Portals

Procedure Use: "titlebar" type, "entity" context, "container" context

The TitleBar script utilizes the following special symbols:

- **text:** (String) Entry: Contains the default text to display ("Choose an Item from the List Below"). Exit: Contains the text to be displayed to the user to prompt selection. The final text may contain encoding.

DESCRIPTION

The TitleBar script is only used within portals that display a choose form where the user can select an item. At the top of the choose form is a title bar that contains a prompt, directing the user to select an item from the list presented. This script allows an author to specify the exact text to be displayed at the top of the selection table, making it possible to remind the user of the context being selected or how many selections remain.

NOTE! The TitleBar script is read-only. Within this script, virtually all aspects of the structural hierarchy can be accessed, but nothing can be changed.

EXAMPLE

When displaying selections where the user can select multiple items, it's helpful to inform the user how many choices remain. This can be easily achieved by a script like the one below.

```
@text = "Add a Special Ablity - " & hero.child[resAbility].field[resSummary].text
```

HEADERTITLE SCRIPT

TECHNICAL DETAILS

Initial	**Context:**	Pick
Alternate	**Context:**	None
Fields	**Finalized?**	Yes
Where	**Used:**	Portals

Procedure Use: "label" type, "info" context, "pick" context

The HeaderTitle script utilizes the following special symbols:

- **text:** (String) Entry: Contains the empty string. Exit: Contains the text to be displayed to the user as the title above the table. The final text may contain encoding.

DESCRIPTION

The HeaderTitle script is essentially a special-purpose Label script used in conjunction with tables. This script allows you to synthesize the text to be displayed as a simple header for a table, without all the work of defining a custom template. It display a single line of text, centered within the header region above the table.

The initial context for the script is dictated by the "headerpick" attribute for the table. The engine will locate a pick within the container that is derived from the specified thing, and that will be the starting script context. Aside from that, the HeaderTitle script is a standard Label script.

When used for tables shown on the screen, the header text is centered within the header region and defaults to using a 10-point Arial font in soft white with a bold style. When used within sheet output, the text is centered in the header region with a solid, light-grey background, and it uses a 13-point Arial font in black with a bold style.

NOTE! The HeaderTitle script is read-only. Within this script, virtually all aspects of the structural hierarchy can be accessed, but nothing can be changed.

EXAMPLE

The Skeleton files make extensive use of this mechanism for putting a suitable title above each table. For example, above the table for selecting Special Abilities, the header shows an appropriate title that includes how many selections remain.

```
@text = "Special Ablities: " & hero.child[resAbility].field[resSummary].text
```

ADDITEM SCRIPT

TECHNICAL DETAILS

Initial	**Context:**	Pick
Alternate	**Context:**	None
Fields	**Finalized?**	Yes
Where	**Used:**	Portals

Procedure Use: "label" type, "info" context, "pick" context

The AddItem script utilizes the following special symbols:

- **text:** (String) Entry: Contains the empty string. Exit: Contains the text to be displayed to the user as the "add" item at the bottom of the table. The final text may contain encoding.

DESCRIPTION

The AddItem script is essentially a special-purpose Label script used in conjunction with tables. This script allows you to synthesize the text to be displayed within a simple "add item" at the bottom of a table, without all the work of defining a custom template. The "add item" for a table enables the user to click within it to add a new item to the table.

The initial context for the script is dictated by the "addpick" attribute for the table. The engine will locate a pick within the container that is derived from the specified thing, and that will be the starting script context. Aside from that, the AddItem script is a standard Label script.

The simple "add item" provided via the mechanism displays a single line of text, centered within the item at the bottom of the table. The text is centered within the item and defaults to using a 10-point Arial font in soft white with a bold style.

NOTE! The AddItem script is read-only. Within this script, virtually all aspects of the structural hierarchy can be accessed, but nothing can be changed.

EXAMPLE

The Skeleton files make extensive use of this mechanism for putting a tailored item at the bottom of each table that prompts the user to add the appropriate kind of object. You can also color-code the text to indicate whether items need to be added or too many items have been added. For example, at the bottom of the table for selecting Special Abilities, the "add item" shows an appropriate message and highlights it based on how many selections remain.

```
~set the color based on whether the proper number of slots are allocated
if (field[resLeft].value = 0) then
  @text = "{text a0a0a0}"
elseif (field[resLeft].value < 0) then
  @text = "{text ff0000}"
  endif
@text &= "Add New Special Ability"
```

CHOSEN SCRIPT

TECHNICAL DETAILS

Initial	**Context:**	Pick
Alternate	**Context:**	None
Fields	**Finalized?**	Yes
Where	**Used:**	Portals

Procedure Use: None

The Chosen script utilizes the following special symbols:

- **text:** (String) Entry: Contains the default text to display ("-Please Select-"). Exit: Contains the text to be displayed to the user as the chosen selection. The final text may contain encoding.
- **ispick:** (Number) Entry: Indicates whether the chooser contains a selection or not. Exit: Ignored.

DESCRIPTION

The Chosen script is essentially a special-purpose Label script used in conjunction with choosers. This script allows you to synthesize the text to be displayed as the current selection within a chooser. If nothing has been selected yet, you can prompt the user to do so, and you can use color-coding to indicate errors with the current selection.

The initial context for the script is the currently selected item. If nothing is selected, the initial context with be invalid, so you're limited to displaying something simple. Aside from that, the Chosen script is a standard Label script.

NOTE! The Chosen script is read-only. Within this script, virtually all aspects of the structural hierarchy can be accessed, but nothing can be changed.

EXAMPLE

Choosers provide an excellent solution when the user must choose exactly one option from an assortment. An excellent example is race, where a character must be assigned a single race. The code below shows the Chosen script for the race chooser within the Skeleton files, including color highlighting when no selection has yet been made.

```
if (@ispick = 0) then
   @text = "{text ff0000}Select Race"
 else
   @text = "Race: " & field[name].text
   endif
```

LEADSUMMARY SCRIPT

TECHNICAL DETAILS

Initial		**Context:**	Hero
Alternate		**Context:**	None
Fields		**Finalized?**	Yes
Where	**Used:**	Definition	File

Procedure Use: "container" context

The LeadSummary script utilizes the following special symbols:

- **text:** (String) Entry: Contains the text "No Summary Provided". Exit: Contains the text to be output as the summary for the lead actor. The final text may contain encoding.

DESCRIPTION

When a user imports characters from a saved portfolio, a summary is shown next to the name. This summary provides basic details about the character that are unique to each game system. The LeadSummary script is used to synthesize the summary. The script is invoked whenever a portfolio is saved so that the summary can be stored in the portfolio and subsequently retrieved for display during import.

The LeadSummary script starts with the leading actor as its initial context. You can then cull whatever information you need from the actor for inclusion within the summary.

NOTE! The LeadSummary script is read-only. Within this script, virtually all aspects of the structural hierarchy can be accessed, but nothing can be changed.

EXAMPLE

In the Mutants & Masterminds game system, each character is shown with its current power points and power level. The script below shows how this is done.

```
~Show our total PP and PL
 @text = herofield[SpentPP].value & " PP"
 @text &= ", PL " & herofield[CurrentPL].value
```

SYNTHESIZE SCRIPT

TECHNICAL DETAILS

Initial		**Context:**	Hero
Alternate		**Context:**	None
Fields		**Finalized?**	Yes
Where		**Used:**	Dossiers

Procedure Use: "synthesize" type, "container" context

The Synthesize script utilizes the following special symbols:

- **newline:** (String) Entry: Contains the appropriate text to begin a new line of text within the output being synthesized (i.e. insert a line break). Exit: Ignored.
- **boldon:** (String) Entry: Contains the appropriate text to turn on output of bold text. Exit: Ignored.
- **boldoff:** (String) Entry: Contains the appropriate text to turn off output of bold text. Exit: Ignored.
- **italicson:** (String) Entry: Contains the appropriate text to turn on output of italic text. Exit: Ignored.
- **italicsoff:** (String) Entry: Contains the appropriate text to turn off output of italic text. Exit: Ignored.
- **separator:** (String) Entry: Contains the appropriate text to insert a suitable horizontal separator. In HTML output, this is the "<hr>" element, and elsewhere it's a line consisting of a string of dashes. Exit: Ignored.

DESCRIPTION

The Synthesize script is utilized when generating text output for a character dossier. A classic example is the creation of statblock output. When the user requests text output, he will also specify the style to be used when synthesizing the output (e.g. HTML, BBCode, or plain text). Before the script is invoked, HL will setup a number of appropriate mechanisms for that output style, which are provided via the various special symbols. For example, in HTML output, the "@boldon" special symbol will map to "<b>", while it will be "[b]" for BBCode output and do nothing for plain text output. This makes it possible for you to write a single Synthesize script that will generate output properly for each different style, without having to do any special handling within the script.

Unlike other scripts, the Synthesize script does not use a "@text" special symbol. Since the volume of output in some cases will be significant, using the standard approach simply isn't practical or efficient. Instead, the output is generated in sequential fashion, and the

"append" language statement is used to accomplish this (see details). The "append" statement allows you to systematically construct the output, one section at a time. Once something is output, you can forget about it and let HL handle it.

If you need to do special formatting within a Synthesize script that is based on the output style, you can. You can find out the style by using the the "isdossierstyle" script target reference from within the State script context.

The Synthesize script starts with the actor to be output as its initial context. You can then cull whatever information you need from the actor for inclusion within the output.

NOTE! The Synthesize script is read-only. Within this script, virtually all aspects of the structural hierarchy can be accessed, but nothing can be changed.

NOTE! While within a Synthesize script and any procedures that are invoked, a global tag is automatically assigned by HL. The tag belongs to the "dossier" tag group and has the unique id of the dossier being output. This allows the script code to base its behavior on the actual dossier that the user is outputting.

EXAMPLE

Every statblock starts with the character's name and then continues with other appropriate information that depends on the game system. The example below shows the start of a Synthesize script that includes the name, race, and age of the character.

```
var txt as string

~start by getting our name
 if (empty(hero.actorname) = 0) then
   txt = hero.actorname
 else
   txt = "Unnamed Character"
   endif

~output our name
 append @boldon & "Name: " & @boldoff & txt & @newline

~output any race
 txt = hero.firstchild["Race.?"].field[name].text
 if (empty(txt) <> 0) then
   txt = "-none-"
   endif
 append @boldon & "Race: " & @boldoff & txt & @newline

~output age
 append @boldon & "Age: " & @boldoff & hero.child[mscPerson].field[perAge].text & @newline
```

TRIGGER

These scripts are invoked in direct response to user actions, such as merging and splitting stackable gear, controlling combat and turns, etc.

TRIGGER SCRIPT

TECHNICAL DETAILS

Initial	**Context:**	Pick
Alternate	**Context:**	None
Fields	**Finalized?**	Yes
Where	**Used:**	Portals

Procedure Use: "trigger" type, "pick" context

The Trigger script utilizes the following special symbols:

- **None:** There are no special symbols for a Trigger script.

DESCRIPTION

The Trigger script is used in conjunction with the "trigger" action portal, which makes it possible for the user to invoke a one-time operation. The primary use of the Trigger script is for either resetting values (e.g. trackers on the In-Play tab) or applying adjustments to usage pools (e.g. damage tracking). In the latter case, there will typically be an edit portal where the user can enter a value. When the Trigger script is invoked, the value is used when applying the adjustment to the usage pool, and then the field is reset.

When invoked, a Trigger script starts with the pick as its initial context. There is generally no reason to navigate outside the context of the pick, but it is technically allowed. The primary focus will be on the pick and its fields.

EXAMPLE

We'll use the Trigger script associated with the button to reset all damage on the In-Play tab as an example. In this script, the usage pools for damage tracking are reset.

```
~if there is no history to undo, notify the user
 if (hero.usagepool[DmgNet].count = 0) then
   notify "Undo history is empty"
   done
   endif

~empty out both usage pools
 perform hero.usagepool[DmgNet].empty
 perform hero.usagepool[DmgAdjust].empty
```

INTEGRATE SCRIPT

TECHNICAL DETAILS

Initial		**Context:**	Pick
Alternate		**Context:**	None
Fields		**Finalized?**	No
Where	**Used:**	Definition	File

Procedure Use: "integrate" type, "pick" context

The Integrate script utilizes the following special symbols:

- **None:** There are no special symbols for a Integrate script.

DESCRIPTION

The Integrate script serves a very specific purpose. When the user triggers the integration of pending actors into an existing combat within the Tactical Console, this script allows the author to piggyback additional special handling. In general, you should not need to perform any special behaviors, but the mechanism is provided just in case.

When integration is triggered by the user, the Integrate script is applied to each actor after it has been properly integrated into the combat. When invoked, the Integrate script starts with the "actor" pick of an actor as its initial context.

EXAMPLE

In a game system like D&D 4th Edition, there are "encounter" powers, which can be used once per encounter. It might make sense for a game like this to automatically reset the state of each encounter power within the Integrate script. If so, then the corresponding script might look like below.

```
foreach pick in hero where "Power.Encounter"
   eachpick.field[pwrIsUsed].value = 0
   nexteach
```

NEWCOMBAT SCRIPT

TECHNICAL DETAILS

Initial		**Context:**	Pick
Alternate		**Context:**	None
Fields		**Finalized?**	No
Where	**Used:**	Definition	File

Procedure Use: "newcombat" type, "pick" context

The NewCombat script utilizes the following special symbols:

- **isfirst:** (Number) Entry: Non-zero if this script is being invoked for the first actor in the portfolio. Exit: Ignored.

DESCRIPTION

The NewCombat script is invoked whenever the user triggers the start of a new combat within the Tactical Console. Prior to the actual transition into combat mode, this script is applied to each actor. When invoked, the NewCombat script starts with the "actor" pick of an actor in the combat as its initial context. It is invoked for all actors in the combat (non-combatants are ignored), and that is done before the NewTurn or Initiative scripts are invoked. This allows the author to reset state for each actor prior to the new combat.

EXAMPLE

In various game systems, there is the notion of "waiting" or "holding an action". This state will persist for each actor after combat ends, so it needs to be reset at the start of combat.

```
herofield[acAbandon].value = 0
```

ENDCOMBAT SCRIPT

TECHNICAL DETAILS

Initial		**Context:**	Pick
Alternate		**Context:**	None
Fields		**Finalized?**	No
Where	**Used:**	Definition	File

Procedure Use: "endcombat" type, "pick" context

The EndCombat script utilizes the following special symbols:

- **isfirst:** (Number) Entry: Non-zero if this script is being invoked for the first actor in the portfolio. Exit: Ignored.

DESCRIPTION

The EndCombat script is the counterpart of the NewCombat script and is invoked whenever the user triggers the end of a combat within the Tactical Console. When combat is ended, this script is applied to each actor. Like its counterpart, the EndCombat script starts with the "actor" pick of an actor in the combat as its initial context. It is invoked for all actors in the combat (non-combatants are ignored). This allows the author to reset state for each actor after combat completes.

EXAMPLE

In various game systems, there is the notion of "waiting" or "holding an action". This state will persist for each actor after combat ends, so it can be reset at the end of combat instead of at the start.

```
herofield[acAbandon].value = 0
```

NEWTURN SCRIPT

TECHNICAL DETAILS

Initial		**Context:**	Pick
Alternate		**Context:**	None
Fields		**Finalized?**	No
Where	**Used:**	Definition	File

Procedure Use: "newturn" type, "pick" context

The NewTurn script utilizes the following special symbols:

- **isfirst:** (Number) Entry: Non-zero if this script is being invoked for the first actor in the portfolio. Exit: Ignored.

DESCRIPTION

The NewTurn script is invoked whenever the user triggers the start of a new combat turn within the Tactical Console. Prior to the start of the new turn, this script is applied to each actor. When invoked, the NewTurn script starts with the "actor" pick of an actor in the combat as its initial context. It is invoked for all actors in the combat (non-combatants are ignored), being performed after any NewCombat script and before any Initiative script. This allows the author to reset state for each actor prior to each new combat turn.

EXAMPLE

If a game system has state for each actor that needs to be reset at the start of each new combat turn, that can be accomplished via the NewTurn script.

```
herofield[acState].value = 0
```

INITIATIVE SCRIPT

TECHNICAL DETAILS

Initial Context:			Pick
Alternate Context:			None
Fields Finalized?			No
Where Used:		Definition	File

Procedure Use: None

The Initiative script utilizes the following special symbols:

- **initiative:** (Number) Entry: The previous initiative value for the actor or zero if not yet specified. Exit: Calculated initiative value to use for the actor.
- **tiebreaker:** (Number) Entry: The previous tie-breaker value for the actor or zero if not yet specified. tiebreaker Exit: Calculated tie-breaker initiative value to use for the actor.

DESCRIPTION

The Initiative script is used to generate appropriate initiative values for each actor. It is invoked whenever a new initiative value is needed, as determined by the configuration settings in the definition file. It is always invoked after any NewCombat or NewTurn script is invoked.

There are two separate initiative values that must be generated. The first is the standard initiative value used for the game system. Once it is calculated, it should be assigned to the "initiative" special symbol. The second initiative value is the tie-breaker to be used if the two actors have the exact same primary initiative score. For example, in the d20 System, the dexterity bonus is used as a tie- breaker. This value should be assigned to the "tiebreaker" special symbol.

When invoked, the Initiative script starts with the "actor" pick of an actor in the combat as its initial context. From there, you can access whatever facets of the character are necessary to properly calculate the initiative scores.

EXAMPLE

In the d20 System, the initiative score is a d20 roll plus the initiative bonus, while the dexterity bonus is used as the tie-breaker. This yields a script that looks like the following.

```
~Initiative is a d20 roll (0 to 19 + 1) plus initiative bonus
@initiative = random(20) + 1 + hero.child[trInit].field[trtFinal].value
~Tie-breaker is the dexterity bonus
@tiebreaker = hero.child[attrDex].field[trtFinal].value
```

MERGE SCRIPT

TECHNICAL DETAILS

Initial Context:		Pick
Alternate Context:		Pick
Fields Finalized?		Yes
Where Used:		Components

Procedure Use: None

The Merge script utilizes the following special symbols:

- **None:** There are no special symbols for the Merge script.

DESCRIPTION

The Merge script is used to handle the merging of two separate picks into one via stacking behavior. It is invoked whenever the user performs a merge operation of stackable gear. The goal of the script is to allow authors to properly reconcile whatever fields are necessary for an accurate merging of the two picks.

The Merge script is defined for a component. As such, it is inherited into every thing that derived from that component. Each script is intended to properly merge the fields within that component and nothing else. This ensures that a thing derived from multiple components, each with their own Merge script, will perform appropriate merge behavior without incident.

When invoked, the Merge script starts out with a pick as its initial context. However, there are two picks involved in the merge operation. When a merge takes place, there is the pick that is assimilating the other (e.g. increasing its quantity) and the pick that is ultimately deleted once the merge completes. The initial context is the pick that is being assimilated into. The second pick is made available via an alternate pick context, which can be accessed via the "altpick" context. This makes it possible to assimilate one pick into the other, regardless of what fields or behaviors are involved.

EXAMPLE
The sample Merge script below adds the pertinent quantities of the pick being merged into the pick that is being assimilated into.

```
field[trkMax].value += altpick.field[trkMax].value
field[trkUser].value += altpick.field[trkUser].value
```

SPLIT SCRIPT

TECHNICAL DETAILS
Initial Context:			Pick
Alternate Context:			Pick
Fields Finalized?			Yes
Where Used:			Components
Procedure Use: None			

The Split script utilizes the following special symbols:

- **None:** There are no special symbols for the Split script.

DESCRIPTION
The Split script handles the splitting of one piece of gear into two separate picks via stacking behavior. It is invoked whenever the user performs a split operation of stackable gear. The goal of the script is to allow authors to properly reconcile whatever fields are necessary for an accurate splitting of one pick into two distinct picks.

The Split script is defined for a component. As such, it is inherited into every thing that derived from that component. Each script is intended to properly split the fields within that component and nothing else. This ensures that a thing derived from multiple components, each with their own Split script, will perform appropriate split behavior without incident.

When invoked, the Split script starts out with a pick as its initial context. However, there are two picks involved in the split operation. When a split takes place, there is the initial pick that is being split (e.g. decreasing its quantity) and the new pick that is ultimately created through the split process. The initial context is the existing pick that is being split. The second pick is made available via an alternate pick context, which can be accessed via the "altpick" context. This makes it possible to split one pick into two, regardless of what fields or behaviors are involved.

When splitting picks, the engine will automatically update the "stackQty" field of both picks with the appropriate values. Therefore, if a pick with a quantity of 20 is split, with a new quantity of 7 specified, the engine will setup the pick being split with a "stackQty" field of 13 and the new pick with a "stackQty" of 7. You can then use these field values to accurately adjust everything else for each pick.

EXAMPLE
The sample Split script below splits one pick into two and properly adjusts all user-tracking details to accurately reflect the impact of the change.

```
~save the quantity of ammo that still remains unused
 var user as number
 user = field[trkUser].value

~update the new "max" values for both picks based on the new stack quantity
 field[trkMax].value = field[stackQty].value
 altpick.field[trkMax].value -= altpick.field[stackQty].value

~the user value for the first pick is the quantity of ammo still unused,
~subject to the maximum size for the pick
 field[trkUser].value = minimum(user,field[trkMax].value)

~subtract the quantity allocated to the first pick from what's now left
 user -= field[trkUser].value

~if we have any unused ammo left to assign to the other pick, assign it
 if (user <= 0) then
   altpick.field[trkUser].value = 0
 else
   altpick.field[trkUser].value = user
   endif
```

CHANGE SCRIPT

TECHNICAL DETAILS

Initial Context:	Container	**Alternate Context:**	None
Fields Finalized?	Yes	**Where Used:**	Portals

Procedure Use: "container" context

The Change script utilizes the following special symbols:

- **None:** There are no special symbols for the Change script.

DESCRIPTION

The Change script is utilized within chooser and menu portals. The purpose of the script is to enact special handling whenever the contents of the portal are changed.

It is rare that you'll need to use this mechanism, because the impact of a selection will usually be handled via Eval scripts during the next evaluation cycle. The key exception to this is when the user is customizing a gizmo within the form for that gizmo. If the influence of the selection is required before the gizmo changes are finally saved, a Change script is needed to apply the necessary effects.

As an example, consider the World of Darkness data files. When a character spends XP during advancement, the XP cost for each advancement varies based on a numerous conditions. The Change script is utilized to properly calculate the XP cost for the selected advancement, which can then be displayed to the user before the advancement is officially added.

When invoked, the Change script starts out with a container as its initial context. This container is the one that the portal is associated with. For example, a chooser portal will reside within a layout, which is within a panel or form. The container associated with the panel or form is the initial context.

EXAMPLE

The Change script below shows how the calculation of XP cost is performed for the World of Darkness game system. This example is edited down to show the core behaviors, as the real script handles additional factors.

```
~get the cost of each dot for the ability
 var eachdot as number
 eachdot = firstchild["Advance.Gizmo"].field[idotcost].value

~determine the tagexpr with which to identify the pick we're interested in
 var tagexpr as string
 tagexpr = "component.CanAdvance & Advance." & firstchild["Advance.Gizmo"].idstring

~if there is a half-price discount for the ability, incorporate that
 if (hero.tagsearch["HalfPrice." & firstchild["Advance.Gizmo"].idstring] > 0) then
   eachdot = eachdot / 2    eachdot = round(eachdot,0,1)
   endif

~if there is a double-cost penalty for the ability, incorporate that
 if (hero.tagsearch["DoubleCost." & firstchild["Advance.Gizmo"].idstring] > 0) then
   eachdot *= 2    endif

~get the previous dot level
 current = hero.firstchild[tagexpr].field[iprevlevel].value

~get the current dot level
 nextlevel = hero.firstchild[tagexpr].field[ilevel].value

~determine the XP cost to improve to the next dot level
var xp as number
 var dotcount as number
 var dotcost as number
 dotcount = nextlevel
 call dotcost  xp = dotcost
 dotcount = current
 call dotcost  xp -= dotcost
 xp = xp * eachdot

~save out the calculated XP cost
 child[advDetails].field[ixpcalc].value = xp
```

```
child[advDetails].field[inewdots].value = nextlevel
```

TRANSACTION

These scripts are associated with the buying and selling of equipment.

TRANSACTSETUP SCRIPT

TECHNICAL DETAILS

Initial Context:	Pick	or	Thing
Alternate Context:			None
Fields Finalized?			Yes
Where Used:			Components

Procedure Use: "xactsetup" type, "transact" context, "pick" context

The TransactSetup script utilizes the following special symbols:

- **isbuy:** (Number) Entry: Indicates whether the transaction is buy (non-zero) or sell (zero). Exit: Ignored.
- **ispick:** (Number) Entry: Indicates whether the context is a pick (non-zero) or a thing (zero). Exit: Ignored.
- **special:** (Number) Entry: Value specified with the definition of the portal to allow different behaviors based on usage. Exit: Ignored.

DESCRIPTION

The TransactSetup script is invoked at the beginning of a transaction, whether it be a buy or sell operation. When invoked, the initial context is either the thing being purchased or the pick being sold. The script can also access the separate "transaction" context to manipulate the transaction pick. In fact, the role of the TransactSetup script is to appropriately configure the transaction pick so that everything is properly presented and handled for the user.

The engine will automatically setup the transaction pick with a few pieces of information. It's up to you to setup the remaining details for how you want the transaction to be handled. The automatic behaviors of the engine differs for buying and selling operations, so both are outlined separately below.

Setup logic for a "buy" transaction:

1. The "xactName" field is set to the name of the thing being purchased
2. The "xactLimit" field is set to no limit
3. The "xactQty" field is set to the lot size for the thing being purchased
4. The TransactSetup script is invoked

Setup logic for a "sell" transaction:

1. The "xactName" field is set to the name of the thing being purchased
2. The "xactLimit" field is set to the total quantity possessed for the pick
3. The "xactQty" field is set to the total quantity possessed
4. The TransactSetup script is invoked

Within the TransactSetup script, the "xactEach" field must always be set to the proper unit cost for the item being purchased or sold. Other fields may be setup as you deem appropriate to the transaction requirements.

Please see the separate documentation for further details on using transactions.

EXAMPLE

The TransactSetup script below shows the behavior of the Sample data files for handling gear transactions. The "each" cost is setup to the cost of the gear, the actual amount paid is set to zero so that default handling is used, and the gear is identified as a holder if appropriate.

```
~start by assuming our unit cost is the cost of one item
 var cost as number
 cost = field[grCost].value

~setup the unit cost for the item
 hero.transact.field[xactEach].value = cost

~zero out the cash amount to be paid (implying use of the standard cost)
 hero.transact.field[xactCash].value = 0

~if the item is gear, setup whether the item holds other gear
 if (isgear <> 0) then
   hero.transact.field[xactHolder].value = gearcount
```

```
endif
```

TRANSACTBUY SCRIPT

TECHNICAL DETAILS

Initial	**Context:**	Thing
Alternate	**Context:**	None
Fields	**Finalized?**	Yes
Where	**Used:**	Components

Procedure Use: "xactbuy" type, "transact" context, "pick" context

The TransactBuy script utilizes the following special symbols:

- **reject:** (String) Entry: The empty string to indicate success. Exit: If non-empty, indicates the transaction was rejected, with the contents being shown to the user as the explanation for the failure.
- **special:** (Number) Entry: Value specified with the definition of the portal to allow different behaviors based on usage. Exit: Ignored.

DESCRIPTION

The TransactBuy script is invoked at the completion of a "buy" transaction. The purpose of the script is to appropriately complete the purchase. It achieves this by retrieving the details of the transaction from the special "transaction" pick. Once the information is retrieved, whatever actions comprise the purchase can be performed, such as subtracting the cost of the purchase from the cash possessed by the character.

When invoked, the initial context is the thing being purchased. However, this script will primarily be interested in the contents of the "transaction" pick, which can be accessed via the "transaction" context. The transaction pick will contain all of the particulars specified by the user for the purchase.

The contents of the transaction pick will depend entirely on the "buy" template utilized. Whatever portals are presented within the buy template will dictate the values of fields within the transaction pick.

The engine will retrieve the final quantity purchased from the "xactQty" field of the transaction pick. The value of this field will dictate the actual quantity assigned to the new pick that is purchased.

Please see the separate documentation for further details on using transactions.

EXAMPLE

The TransactBuy script below shows the behavior of the Sample data files for handling gear transactions. If the gear is free, the transaction is completed without any changes to the character's cash. Otherwise, the appropriate cash is deducted from the proper usage pool, with a rejection error being reported if there is insufficient cash.

```
~if we're buying for free, no cash should be touched, so get out
 if (hero.transact.field[xactIsFree].value <> 0) then
   done
   endif

~get the cash amount specified by the user
 var cash as number
 cash = hero.transact.field[xactCash].value

~if no cash amount was given, get the standard total cost for the purchase
 if (cash = 0) then
   cash = hero.transact.field[xactQty].value * hero.transact.field[xactEach].value
   endif

~if we don't have enough cash to make the purchase, reject the transaction
 if (cash > herofield[acCashNet].value) then
   @reject = "You lack sufficient cash to purchase the item."
   done
   endif

~subtract the amount from the current pool of cash
 perform hero.usagepool[TotalCash].adjust[-cash]
```

TRANSACTSELL SCRIPT

TECHNICAL DETAILS

Initial		**Context:**	Pick
Alternate		**Context:**	None
Fields		**Finalized?**	Yes
Where		**Used:**	Components

Procedure Use: "xactsell" type, "transact" context, "pick" context

The TransactSell script utilizes the following special symbols:

- **reject:** (String) Entry: The empty string to indicate success. Exit: If non-empty, indicates the transaction was rejected, with the contents being shown to the user as the explanation for the failure.
- **special:** (Number) Entry: Value specified with the definition of the portal to allow different behaviors based on usage. Exit: Ignored.

DESCRIPTION

The TransactSell script is invoked at the completion of a "sell" transaction. The purpose of the script is to appropriately complete the sale. It achieves this by retrieving the details of the transaction from the special "transaction" pick. Once the information is retrieved, whatever actions comprise the sale can be performed, such as adding the proceeds of the sale to the cash possessed by the character.

When invoked, the initial context is the pick that is being sold. However, this script will primarily be interested in the contents of the "transaction" pick, which can be accessed via the "transaction" context. The transaction pick will contain all of the particulars specified by the user for the sale.

The contents of the transaction pick will depend entirely on the "sell" template utilized. Whatever portals are presented within the sell template will dictate the values of fields within the transaction pick.

The engine will retrieve the final quantity sold from the "xactQty" field of the transaction pick. The value of this field will dictate the actual quantity deducted from the pick that is sold. If the entire quantity of the pick is sold, the pick is deleted.

Please see the separate documentation for further details on using transactions.

EXAMPLE

The TransactSell script below shows the behavior of the Sample data files for handling gear transactions. The cash value specified by the user is retrieved and added back to the usage pool the manages the character's cash.

```
~get the cash amount specified by the user
 var cash as number
 cash = hero.transact.field[xactCash].value

~add the amount to the current pool of cash
 perform hero.usagepool[TotalCash].adjust[cash]
```

MODE TRANSITION

These scripts associated with the transition into and out of advancement mode.

CANADVANCE SCRIPT

TECHNICAL DETAILS

Initial		**Context:**		Hero
Alternate		**Context:**		None
Fields		**Finalized?**		Yes
Where	**Used:**		Definition	File

Procedure Use: None

The CanAdvance script utilizes the following special symbols:

- **message:** (String) Entry: The empty string to indicate the character can transition to advancement mode. Exit: If non-empty, indicates the character cannot transition to advancement mode, with the contents being shown to the user as the explanation of what must still be completed for the character. The text may contain encoding.

DESCRIPTION

The CanAdvance script is only utilized when the formalized advancement mechanism is enabled for the game system. This script allows the author to verify that the minimum configuration has been completed for a character before allowing the user to switch from creation mode to advancement mode.

The CanAdvance script is invoked when the user attempts to switch to advancement mode via the menu. When the user tries to make the change, the script is triggered, at which point the script verifies that the necessary creation steps have been completed for the character. If any are not satisfied, an error message is displayed to the user and the user is not allowed to transition until the creation process is completed.

It is the responsibility of the script to synthesize the message shown to the user that reports the creation steps that have not been satisfactorily completed. If a problem is detected in the script, a suitable error message should be appended to the "message" special symbol. If multiple errors occur, the message should contain a list of them, with each message on a new line. If there are no errors, then the special symbol should be the empty string, and that tells HL that the character is ready to transition to advancement mode.

When invoked, a CanAdvance script starts with the actor as its initial context. You are free to check any aspect of the character when verifying that the transition to advancement mode is allowed.

NOTE! Don't be too picky with the CanAdvance script. Remember that every gaming group has its own house rules that will modify the basic game rules. As such, it's important to use validation rules to trap most errors instead of requiring the user to obey all rules. The CanAdvance script should only be used to enforce details that should always be required and/or that impact your ability to write quality data files.

EXAMPLE

The CanAdvance script below shows the behavior of the Sample data files for handling the advancement test. The script sets up a bullet character to put at the start of each error and then performs the tests. If any test fails, a suitable error message is appended to the message.

```
var bullet as string
bullet = "{bmp bullet_red}{horz 4}"

~perform tests to assure starting resources have been assigned
 if (#resleft[resCP] <> 0) then
   @message = @message & bullet & "Character points must be assigned for the character.\n"
   endif
 if (#resleft[resAbility] <> 0) then
   @message = @message & bullet & "Ability slots must be assigned for the character.\n"
   endif
```

TRANSITION SCRIPT

TECHNICAL DETAILS

Initial Context:			Hero
Alternate Context:			None
Fields Finalized?			Yes
Where Used:		Definition	File

Procedure Use: None

The Transition script utilizes the following special symbols:

- **message:** (String) Entry: The empty string to indicate that no transition message should be shown to the user. Exit: If non-empty, specifies the transition message to be shown to the user as a reminder. The text may contain encoding.

DESCRIPTION

The Transition script is only utilized when the formalized advancement mechanism is enabled for the game system. This script allows the author to display a message to the user whenever the user successfully switches from creation mode to advancement mode, or vice versa. In general, the message is simply a reminder about the implications of the transition, as well as how to switch back. If you want to suppress the message, simply use an empty message string (or omit the script entirely). The message starts out centered, although you can change the behavior.

When invoked, a Transition script starts with the actor as its initial context. You are free to refer to any aspect of the character, although doing so is rarely necessary. The primary detail you'll want to check is whether the character is transitioning to creation mode or advancement mode, which can be determined via the "state" context.

EXAMPLE

The Transition script below shows the standard behavior you'll want to include in your data files. Depending on the new mode, an appropriate message is displayed.

```
if (state.iscreate <> 0) then
   @message = "{b}{text ffff00}Creation Phase{text 010101}{/b}"
   @message &= "\n\n"
```

```
  @message &= "{align left}You have unlocked your character!"
else
  @message = "{b}{text ffff00}Advancement Phase{text 010101}{/b}"
  @message &= "\n\n"
  @message &= "{align left}You have locked your character creation traits. "
  endif
```

RELEASE CHANGES

These scripts are used to accommodate changes between data file releases and potential loading errors of portfolios.

LOADERROR SCRIPT

LOADFIXUP SCRIPT

DEFINITION FILE REFERENCE

Each game system has a single definition file that describes the fundamental, non-changing characteristics of the game. HL uses the definition file to configure itself for each game system, reading in the definition file before any other files. All of the contents of the other data files are given meaningful interpretation by the contents of the definition file.

The definition file must have the file extension ".def". By convention, the file is always named "definition.def" so that it can always be readily identified.

This section outlines the structure and mechanics for writing a definition file.

IMPORTANT! This section utilizes critical notational conventions that should be reviewed.

STRUCTURAL COMPOSITION

The overall file structure is that of a standard XML file. The file must start with an XML version element in the form: "<?xml version="1.0"?>". Following this, the top-level XML element must be a "document" and it must have a "signature" attribute containing the explicit value "Hero Lab Definition".

The following table defines the attributes for a "document" element.

- **signature:** Text. Must be the value "Hero Lab Definition".

Within the document element, every definition file possesses the following child elements, appearing in the sequence given and with the names specified.

- **game:** A single "game" element must appear as defined by the given link. This element contains basic game system information.
- **author:** An optional "author" element may appear as defined by the given link. This element contains information about the author of the data files.
- **release:** A single "release" element must appear as defined by the given link. This element provides details about the current release of the data files.
- **structure:** A single "structure" element must appear as defined by the given link. This element specifies structural details about the game system and its behavior.
- **behavior:** A single "behavior" element must appear as defined by the given link. This element details behavioral aspects for the game system and data files.
- **scriptmacro:** Zero or more "scriptmacro" elements may appear as defined by the given link. This element defines macros for use within scripts.
- **advancement:** An optional "advancement" element may appear as defined by the given link. This element contains details regarding the advancement works within the data files.
- **phase:** Zero or more "phase" elements may appear as defined by the given link. This element dictates the set of evaluation phases that will exist for the game system.

DEFINITION FILE ELEMENTS

GAME ELEMENT (DATA)

THE "GAME" ELEMENT

The game system information section defines the name of the game system, the manufacturer, and copyright information. The XML element name is "game" and the complete list of attributes is below.

- **game:** Text. Name of the game system. Maximum length is 50 characters.
- **publisher:** Text. Name of the manufacturer or publisher of the game system. Maximum length is 50 characters.
- **website:** Text. URL of the publisher's web-site. Maximum length is 100 characters.
- **copyright:** (Optional) Text. Appropriate copyright and trademark declaration for the game system and data files. Maximum length is 250 characters. Default: Empty.
- **legaloutput:** Text. This is shortened legal text that is inserted at the bottom of all character sheet output. You must provide appropriate copyright and trademark declarations.
- **manualroot:** (Optional) Text. Specifies the name of the file to be used as the User Manual for the game system, which is accessible via the Configure Hero form and the Help menu within HL. This should generally be an HTML file. Default: "docs\manual.htm".
- **editorroot:** (Optional) Text. Specifies the name of the file used as the manual for working with the Editor when you data files are loaded. Within the Editor, this manual is accessible via the Help menu. This should generally be an HTML file. Default: "docs\editor.htm".

EXAMPLE

The following example demonstrates what a "game" element might look like. All default values are assumed for omitted optional attributes.

```
<game
  name="Sample Game Systems"
 publisher="Game Company"
  website="www.gamecompany.com"
  copyright="Copyright 2008 by Game Company. Game XYZ is a trademark of Game Company."
  legaloutput="Copyright 2008 by Game Company.">
  </game>
```

AUTHOR ELEMENT (DATA)

THE "AUTHOR" ELEMENT

The author information section contains details about the author of the data files and how best to contain the author regarding the data files. The XML element name used is "author" and the complete list of attributes is below.

- **author:** Text. Name of the author. Maximum length is 50 characters.
- **email:** (Optional) Text. Email address of the author to send questions and/or bug reports to. Maximum length is 70 characters. Default: Empty.
- **website:** (Optional) Text. URL of the author's web-site where a FAQ and/or useful information can be found. Maximum length is 100 characters. Default: Empty.

EXAMPLE

The following example demonstrates what an "author" element might look like. All default values are assumed for optional attributes.

```
<author
  author="John Q. Author"
  website="www.johnqauthor.com">
  </author>
```

RELEASE ELEMENT (DATA)

THE "RELEASE" ELEMENT

All details pertaining to the current release of the data files can be found here. The XML element name used is "release" and the complete list of attributes is below.

- **major:** Integer. Major version number assigned to the release. Must be between 0-255.
- **minor:** Integer. Minor version number assigned to the release. Must be between 0-255.
- **required:** (Optional) Text. Specifies the minimum version of HL required to utilize these data files. The version is given in the exact same format displayed by the product when the "About Hero Lab" option is selected under the "Help" menu (e.g. "2.3a"). If empty, no requirements are enforced and the data files are assumed to work with all versions of the product. Default: Empty.
- **summary:** (Optional) Text. Arbitrary text used as release summary notes to display to the user. This summary is used by the HLExport tool when packaging up your data files for distribution to other users. No maximum length. Default: Empty.
- **PCDATA:** (Optional) Text. Detailed release notes for the data files can be specified within the PCDATA block of the element. These release notes will be displayed to the user every time the data files are re-compiled and then loaded. This is an ideal place to

inform users about the status of the data files, new capabilities that have been added, and where to get answers to questions about using the data files.

EXAMPLE

The following example demonstrates what a "release" element might look like. All default values are assumed for omitted optional attributes.

```
<release
  major="2"
  minor="1"
 summary="Release summary here"><![CDATA[These are the actual release notes.
  ]]></release>
```

STRUCTURE ELEMENT (DATA)

THE "STRUCTURE" ELEMENT

Every game system will have an assortment of structural details that must be specified, and these are all grouped together within a "structure" element. The complete list of attributes for a structure element is below.

- **folder:** Text. Unique name to be used for the folder in which all data files for this game system are placed. The name may consist only of alphanumeric characters (i.e. letters and digits), with no spaces or punctuation other than "_" and "-" allowed. Maximum length is 25 characters. **WARNING!** The folder name has critical implications and must be chosen carefully.
- **editwidth:** (Optional) Integer. Specifies the fixed width to use for all edit (i.e. tab-based) panels within HL. The value is given in units of pixels. Default: "500".
- **summarymin:** (Optional) Integer. Specifies the minimum width that will be allowed for all summary panels within HL. The value is given in units of pixels. Default: "135".
- **summarymax:** (Optional) Integer. Specifies the maximum width that will be allowed for all summary panels within HL. The value is given in units of pixels. Default: "165".
- **heroterm:** (Optional) Text. Name to be used when referring to a hero for this game system. Maximum length is 25 characters. Default: "hero".
- **thingterm:** (Optional) Text. Name to be used when referring to a thing for this game system. Maximum length is 25 characters. Default: "thing".
- **entityterm:** (Optional) Text. Name to be used when referring to an entity for this game system. Maximum length is 25 characters. Default: "entity".
- **combatturnterm:** (Optional) Text. Name to be used when referring to a combat turn for this game system. Maximum length is 25 characters. Default: "turn".

The "structure" element also possesses child elements that describe additional facets of the game system. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **datetime:** Zero or more "datetime" elements may appear as defined by the given link. This element defines rules for date and time composition.

THE "DATETIME" ELEMENT

The "datetime" element dictates the structure of dates and times for the game system. The sequence in which the "datetime" elements are defined dictates the composition used within HL. By default, the date and time structures for gamespace are assumed to be those used in realspace. The complete list of attributes for a "datetime" element is below.

- **name:** Text. Name to be used for the component of the date or time. Maximum length is 20 characters.
- **digits:** Integer. Specifies the number of digits used to specify the date/time component.
- **istime:** Boolean. Indicates whether this is a component of the game system's time or date.

EXAMPLE

The following example demonstrates what a "structure" element might look like. All default values are assumed for optional attributes. Within this example, the composition of a game date is defined as if it were a date in realspace.

```
<structure
  folder="skeleton"
  editwidth="520"
  combatturnterm="round">
```

```
<datetime name="month" digits="2"/>
<datetime name="day" digits="2"/>
<datetime name="year" digits="4"/>
</structure>
```

BEHAVIOR ELEMENT (DATA)

THE "BEHAVIOR" ELEMENT

Each game system has a variety of behaviors that will need to be defined. These behaviors are collectively specified within a "behavior" element. The complete list of attributes for a "behavior" element is below.

- **defaultname:** (Optional) Text. Default name to be used for all actors that have not been explicitly named by the user. Maximum length is 50 characters. Default: "Unnamed".
- **secondaryphase:** (Optional) Id. Specifies the unique id of the phase to be used as the default for all Secondary tag expressions assigned by tables and choosers. If no explicit phase is assigned for a Secondary tag expression, this one is used. Default: "Setup".
- **secondarypriority:** (Optional) Integer. Specifies the priority to be used as the default for all Secondary tag expressions assigned by tables and choosers. If no explicit priority is assigned for a Secondary tag expression, this one is used. Default: "5000".
- **existencephase:** (Optional) Id. Specifies the unique id of the phase to be used as the default for all Existence tag expressions assigned by tables and choosers. If no explicit phase is assigned for an Existence tag expression, this one is used. Default: "Setup".
- **existencepriority:** (Optional) Integer. Specifies the priority to be used as the default for all Existence tag expressions assigned by tables and choosers. If no explicit priority is assigned for an Existence tag expression, this one is used. Default: "5000".
- **gearphase:** (Optional) Id. Specifies the unique id of the phase during which all built-in gear processing is performed. This includes the calculation of gear weights throughout all gear holders. Default: "Effects".
- **gearpriority:** (Optional) Integer. Specifies the priority at which all built-in gear processing is performed. Default: "5000".
- **mouseinfo:** (Optional) Id. Specifies the unique id of the procedure to automatically invoke as the MouseInfo script within portals. Default: "MouseInfo".
- **description:** (Optional) Id. Specifies the unique id of the procedure to automatically invoke as the Description script within portals. Default: "Descript".
- **initascend:** (Optional) Boolean. Indicates whether initiative order ascends or descends. When it ascends, lower numbers take their actions first, and vice versa. Default: "no".
- **initperturn:** (Optional) Boolean. Indicates whether to re-generate new initiative values at the start of every turn instead of at the start of every combat. Default: "no".
- **initsimultaneous:** (Optional) Boolean. Indicates whether it is valid for two actors to have identical initiatives and act simultaneously. If not, HL will automatically generate a tie-breaker value. Default: "no".
- **initminimum:** (Optional) Boolean. Specifies the minimum value allowed for the initiative value when the user is adjusting the value on the Tactical Console. Default: "-99".
- **initmaximum:** (Optional) Boolean. Specifies the maximum value allowed for the initiative value when the user is adjusting the value on the Tactical Console. Default: "99".
- **interleave:** (Optional) Boolean. Indicates whether the evaluation cycles of all actors within the context of the same lead are interleaved, which governs the behavior of minions and masters. If enabled, masters can influence changes upon their minions and minions can also influence changes upon their masters. If not enabled, masters all fully evaluated before minions, so the influence can only occur in the downward direction. Default: "no".

The "behavior" element also possesses child elements that describe additional facets of the game system. The list of these child elements is below and must appear in the order shown. Click on the links to access the details for each element.

- **diceroller:** A single "diceroller" element must appear as defined by the given link. This element customizes the integrated Dice Roller.
- **leadsummary:** An optional "leadsummary" element may appear as defined by the given link. This element defines the LeadSummary Script. If omitted, default handling is employed.
- **newcombat:** An optional "newcombat" element may appear as defined by the given link. This element defines the NewCombat Script. If omitted, default handling is employed.
- **newturn:** An optional "newturn" element may appear as defined by the given link. This element defines the NewTurn Script. If omitted, default handling is employed.
- **integrate:** An optional "integrate" element may appear as defined by the given link. This element defines the Integrate Script. If omitted, default handling is employed.
- **initiative:** An optional "initiative" element may appear as defined by the given link. This element defines the Initiative Script. If omitted, default handling is employed.

- **initfinalize:** An optional "initfinalize" element may appear as defined by the given link. This element defines the InitFinalize Script. If omitted, default handling is employed.
- **loaderror:** An optional "loaderror" element may appear as defined by the given link. This element defines the LoadError Script. If omitted, default handling is employed.

THE "DICEROLLER" ELEMENT

The "diceroller" element defines characteristics of the integrated Dice Roller mechanisms when used with the game system. The complete list of attributes for a "diceroller" element is below.

- **mode:** (Optional) Set. Specifies the initial dice rolling mode to start out in for the game system. Must be one of the following:
  - **totals:** The dice rolled are added and that total is displayed.
  - **successes:** Each die rolled is compared against the threshold for declaring a success and the number of successes is displayed.
  - Default: "totals".
- **dietype:** Integer. Specifies the default die type to be rolled.
- **quantity:** (Optional) Integer. Specifies the default number of dice to be rolled. Default: "1".
- **success:** (Optional) Integer. Specifies the threshold value at which a success result is achieved. Game systems like World of Darkness and Shadowrun utilizes successes instead of adding the dice results. Default: "1".
- **explode:** (Optional) Integer. Specifies the threshold value at which a success result "explodes" by allowing a re-roll. If a value of zero is given, no explosion occurs. Default: "0".
- **maxexplode:** (Optional) Integer. Specifies the maximum number of times that a single roll may explode. A value of zero indicates that explosion is unlimited. Default: "0".

THE "LEADSUMMARY" ELEMENT

The "leadsummary" element defines the LeadSummary script that is used to synthesize summary information for each actor. This summary is displayed when the user previews the available actors for import into the current portfolio. The complete list of attributes for a "scriptmacro" element is below.

- **PCDATA:** Script. Specifies the code comprising the LeadSummary script.

THE "NEWCOMBAT" ELEMENT

The "newcombat" element defines the NewCombat script that is invoked whenever the user begins a new combat within the Tactical Console. The complete list of attributes for the element is below.

- **PCDATA:** Script. Specifies the code comprising the NewCombat script.

THE "NEWTURN" ELEMENT

The "newturn" element defines the NewTurn script that is invoked whenever the user starts a new turn of combat within the Tactical Console. The complete list of attributes for the element is below.

- **PCDATA:** Script. Specifies the code comprising the NewTurn script.

THE "INTEGRATE" ELEMENT

The "integrate" element defines the Integrate script that is invoked whenever the user integrates pending actors into an existing combat within the Tactical Console. The complete list of attributes for the element is below.

- **PCDATA:** Script. Specifies the code comprising the Integrate script.

THE "INITIATIVE" ELEMENT

The "initiative" element defines the Initiative script that is used to calculate the initiative score for each actor during combat. The complete list of attributes for the element is below.

- **PCDATA:** Script. Specifies the code comprising the Initiative script.

THE "INITFINALIZE" ELEMENT

The "initfinalize" element defines the InitFinalize script that is used to tailor the final displayed text for the initiative score of each actor during combat. The complete list of attributes for the element is below.

- **PCDATA:** Script. Specifies the code comprising the InitFinalize script.

THE "LOADERROR" ELEMENT

The "loaderror" element defines the LoadError script that is used to provide helpful information to the user about changes across releases of your data files and errors that can be safely ignored. The complete list of attributes for the element is below.

- **PCDATA:** Script. Specifies the code comprising the LoadError script.

EXAMPLE

The following example demonstrates what a "behavior" element might look like. All default values are assumed for omitted optional attributes.

```
<behavior
  defaultname="Nobody"
  interleave="yes">
  <diceroller mode="totals" dietype="6" quantity="1"/>

  <leadsummary><![CDATA[
    ~start with the race
    var txt as string
    txt = hero.firstchild["Race.?"].field[name].text
    if (empty(txt) <> 0) then
      txt = "No Race"
      endif
    @text &= txt

    ~append the CP rating of the character
    @text &= " - CP: " & hero.child[resCP].field[resMax].value
    ]]></leadsummary>

  <initiative><![CDATA[
    ~calculate the secondary initiative rating
    @secondary = 0
    ~calculate the primary initiative rating
    @initiative = random(10) + 1
    ]]></initiative>

  </behavior>
```

SCRIPTMACRO ELEMENT (DATA)

THE "SCRIPTMACRO" ELEMENT

The "scriptmacro" element defines a macro for use within the scripting language to conveniently access object details. Each macro specifies a kind of shorthand notation for a lengthy block of script code that can be used in place of the full code and will be treated as if the full code were entered. Script macros are globally defined and may be used within any script. The complete list of attributes for a "scriptmacro" element is below.

- **name:** Text. Name to be used for the macro. May consist solely of alphanumeric characters.
- **result:** Text. The resulting text to be generated when the macro is used. Each parameter is spliced into the result appropriately.
- **param1:** (Optional) Text. Name used for the first parameter provided to macro. Used for splicing the value into the "result" given above.
- **param2:** (Optional) Text. Name used for the second parameter provided to macro. Used for splicing the value into the "result" given above.
- **param3:** (Optional) Text. Name used for the third parameter provided to macro. Used for splicing the value into the "result" given above.
- **param4:** (Optional) Text. Name used for the fourth parameter provided to macro. Used for splicing the value into the "result" given above.
- **param5:** (Optional) Text. Name used for the fifth parameter provided to macro. Used for splicing the value into the "result" given above.

EXAMPLE

The following example demonstrates what a "scriptmacro" element might look like. All default values are assumed for omitted optional attributes.

```
<scriptmacro
  name="trait"
  param1="trait"
  result="hero.child[#trait].field[trtFinal].value">
  </scriptmacro>
```

ADVANCEMENT ELEMENT (DATA)

THE "ADVANCEMENT" ELEMENT

Some game systems require that each new advancement of a character be performed in a rigid sequence. This is because the cost of increasing individual facets of the character go up as the character gets better. When a game system requires structured advancement, the "advancement" element can be defined to specify fundamental behaviors associated with the advancement. The complete list of attributes for an advancement element is below.

- **enable:** (Optional) Boolean. Indicates whether a serialized advancement mechanism is used for the game system. Default: "no".
- **creationterm:** (Optional) Text. Specifies the term presented to the user when referring to the initial mode where a new character is being created. Default: "creation".
- **advancementterm:** (Optional) Text. Specifies the term presented to the user when referring to the post-creation mode where a character's advancement is serialized. Default: "advancement".

The "advancement" element also possesses child elements that describe additional facets of the advancement behavior. The list of these child elements is below and must appear in the order shown. Click on the links to access the details for each element.

- **canadvance:** An optional "canadvance" element may appear as defined by the given link. This element defines the CanAdvance Script. If omitted, default handling is performed.
- **transition:** An optional "transition" element may appear as defined by the given link. This element defines the Transition Script. If omitted, default handling is performed.

THE "CANADVANCE" ELEMENT

The "canadvance" element defines the CanAdvance script that determines whether a character satisfies the initial creation requirements for transitioning into serialized advancement. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the CanAdvance script.

THE "TRANSITION" ELEMENT

The "transition" element defines the Transition script that is invoked when a character transitions between creation and advancement modes. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the Transition script.

EXAMPLE

The following example demonstrates what an "advancement" element might look like. All default values are assumed for omitted optional attributes.

```
<advancement
  enable="yes"
  creationterm="starting"
  advancementterm="improving">
  <canadvance><![CDATA[
    if (#resleft[resCP] <> 0) then
      @message = "Character does not meet requirement."
      endif
    ]]></canadvance>

  <transition><![CDATA[
    if (#iscreate[@newmode] = 1) then
      @message = "Now Creating!"
    else
      @message = "Now Advancing!"
      endif
    ]]></transition>

  </advancement>
```

PHASE ELEMENT (DATA)

THE "PHASE" ELEMENT

Each phase within the evaluation cycle must be individually specified and the sequence in which the phase elements are defined dictates the sequence in which phases are processed during evaluation. The XML element name is "phase" and the complete list of attributes is below.

- **id:** Id. Specifies the unique id assigned to this phase. This id is used to reference the phase throughout the data files.
- **name:** Text. Common name for the phase is displayed in various situations. Maximum length is 25 characters.

- **description:** (Optional) Text. Textual description of the role the phase serves within the game system. Default: Empty.
- **interleave:** (Optional) Boolean. Indicates whether all tasks within this phase are either (a) interleaved across masters and minions or (b) processed hierarchically, with masters always being evaluated before minions. This works the exact same way as the "interleave" attribute within the "behavior" element, except that it only applies to tasks within this specific phase. Default: "yes". **IMPORTANT!** This attribute is only applicable when interleaving is enabled for the game system. It provides a means of forcing individual phases to be non-interleaved, which makes it possible to safely control dependencies of minions upon their masters for enablement conditions and the like. The default is "yes" so that all phases are interleaved by default if the game system behavior is designated as interleaved.

EXAMPLE
The following example demonstrates what a "phase" element might look like. All default values are assumed for omitted optional attributes.

```
<phase
  id="Initialize"
  name="Initialization"
  description="Anything that has to happen before everything else">
  </phase>
```

STRUCTURAL FILE REFERENCE

Structural files are where you'll be defining all of the pieces that provide the underlying framework for the game system. They are loaded immediately after the definition file.

The Kit supports four different structural file types that share the identical contents. They possess the file extensions ".1st", "core", ".str", and ".aug". These files are always loaded in that same order.

This section outlines the structure and mechanics for writing a structural file.

IMPORTANT! This section utilizes critical notational conventions that should be reviewed.

IMPORTANT! Just because you can put numerous different elements in the same file does not mean you should do so. Keeping your data files small and focused will also keep them much more manageable, so break up all the information across files where appropriate. See the Skeleton data files for examples of this.

STRUCTURAL COMPOSITION

The overall file structure is that of a standard XML file. The file must start with an XML version element in the form: "<?xml version="1.0"?>". Following this, the top-level XML element must be a "document" and it must have a "signature" attribute containing the explicit value "Hero Lab Structure".

The following table defines the attributes for a "document" element.

- **signature:** Text. Must be the value "Hero Lab Structure".

Within the document element, every structural file possesses the following child elements, appearing in the sequence given and with the names specified.

- **enmasse:** Zero or more "enmasse" elements may appear as defined by the given link. This element specifies categories of things to automatically add to every actor.
- **bootstrap:** Zero or more "bootstrap" elements may appear as defined by the given link. This element specifies specific things to automatically add to every actor.
- **autoadd:** Zero or more "autoadd" elements may appear as defined by the given link. This element identifies choices that are pre-selected for every actor.
- **usagepool:** Zero or more "usagepool" elements may appear as defined by the given link. This element specifies usage pools for use by the game system.
- **reference:** Zero or more "reference" elements may appear as defined by the given link. This element details references used throughout the data files.
- **group:** Zero or more "group" elements may appear as defined by the given link. This element defines a collection of tag groups and their tags.
- **source:** Zero or more "source" elements may appear as defined by the given link. This element specifies sources that can be configured by the user for each actor.
- **resource:** Zero or more "resource" elements may appear as defined by the given link. This element defines resources like fonts and bitmaps used by styles for visual display.

- **style:** Zero or more "style" elements may appear as defined by the given link. This element specifies an assortment of visual styles that can be used by portals.
- **component:** Zero or more "component" elements may appear as defined by the given link. This element specifies a variety of components that can be used by component sets.
- **compset:** Zero or more "compset" elements may appear as defined by the given link. This element defines various component sets that can be used to define things.
- **entity:** Zero or more "entity" elements may appear as defined by the given link. This element defines entities that can be used within the game system.
- **sortset:** Zero or more "sortset" elements may appear as defined by the given link. This element defines various sort sets that can be used by the game system.

STRUCTRUAL FILE ELEMENTS

ENMASSE ELEMENT (DATA)

THE "ENMASSE" ELEMENT

For every game system, there will be related things that must be added to every actor. These things can be identified through the use of the "enmasse" element. The complete list of attributes for this element is below.

- **PCDATA:** Text. This tag expression identifies the group of related things that should be added to every actor. Every thing that satisfies the tag expression is added.

The "enmasse" element also possesses child elements that pertain to the automatically added things. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **autotag:** A single "autotag" element may appear as defined by the given link. This element specifies a tag that is automatically assigned to each added thing.

EXAMPLE

The following example demonstrates what an "enmasse" element might look like. All default values are assumed for optional attributes.

```
<enmasse><![CDATA[component.Attribute]]>
  <autotag group="TheGroup" tag="TheTag"/>
  </enmasse>
```

AUTOTAG ELEMENT (DATA)

THE "AUTOTAG" ELEMENT

In a variety of situations, you will be able to specify one or more tags that will be automatically assigned to an object when it is added. This will always be controlled by the mechanism that is adding the object, such as bootstrapping a thing or adding a thing via a particular table.

Each tag that is to be automatically assigned is identified through the use of the "autotag" element. The complete list of attributes for this element is below.

- **group:** Id. Specifies the unique id of the tag group within which the assigned tag is defined.
- **tag:** Id. Specifies the unique id of the assigned tag within the above tag group.

EXAMPLE

The following example demonstrates what a "autotag" element might look like. All default values are assumed for optional attributes.

```
<autotag group="TheGroup" tag="TheTag"/>
```

BOOTSTRAP ELEMENT (DATA)

THE "BOOTSTRAP" ELEMENT

In a variety of situations, you will be able to specify individual things that should be automatically added to a container as picks. This process is called bootstrapping and has a diverse set of special rules that must be observed. Each bootstrapped thing is specified through the use of a "bootstrap" element. The complete list of attributes for this element is below.

- **thing:** Id. Specifies the unique id of the thing that is to be automatically added to the container.

- **index:** (Optional) Integer. Specifies a unique index value for this bootstrap entry. Only global bootstraps that add things to all actors require this attribute and it can be omitted in all other situations. Default: "0". **WARNING!** The index is used to uniquely identify this specific bootstrap within saved portfolios. Consequently, it must **never** be changed or re-used within different versions of the data files. If you eliminate a bootstrap entry, simply leave that index value unused, replacing the old "bootstrap" element with a comment explaining why the hole in numbering is left.

The "bootstrap" element also possesses child elements that pertain to the automatically added things. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **match:** An optional "match" element may appear as defined by the given link. This element defines the Match Tag Expression. If omitted, all things are assumed to match and the bootstrap is assigned to all. **IMPORTANT!** This element is only applicable when the bootstrap is defined directly within a component. In all other cases, this element may not be specified.
- **containerreq:** An optional "containerreq" element may appear as defined by the given link. This element defines the conditional requirements that determine whether the bootstrap is made available within the container. The tag expression is applied against the container of the prospective bootstrapped pick. If the container does not satisfy the tag expression, the bootstrapped pick is treated as non-live. **IMPORTANT!** This element is not applicable when the bootstrap is defined within an entity.
- **autotag:** Zero or more "autotag" elements may appear as defined by the given link. This element specifies tags that are automatically assigned to the added thing. **IMPORTANT!** When the bootstrap include a ContainerReq test, the tags are only assigned to the pick when ContainerReq tag expression is satisfied.
- **assignval:** Zero or more "assignval" elements may appear as defined by the given link. This element specifies modified field values that are automatically assigned to the added thing.

THE "MATCH" ELEMENT

The "match" element defines the Match tag expression that determines whether a particular thing receives the specified bootstrap. The tag expression is applied against each thing derived from the component, and the bootstrap is only assigned to things that satisfy the tag expression. The complete list of attributes for this element is below.

- **PCDATA:** TagExpr. Specifies the code comprising the Match tag expression.

THE "ASSIGNVAL" ELEMENT

All of the initial field values for a pick are dictated by the thing. However, there are situation where you'll want to dictate custom field values when bootstrapping a thing. For example, if you have a "Fly" ability, there may be different flying speeds. The "Fly" ability will have its initial speed, but each situation where you bootstrap the ability may call for a different speed.

To handle this situation, the "assignval" element allows you to designate the initial value to be used for a field within a bootstrapped pick. The complete list of attributes for this element is below.

- **field:** Id. Specifies the unique id of the field for which the value is being assigned.
- **value:** Text. Specifies the new starting value to be assigned to the field. If the field is text-based, the value is simply assigned, although it may be truncated if it exceeds the defined maximum length for the field. If the field is value-based, the text is converted to a floating point value and assigned.
- **behavior:** (Optional) Set. Designates the assignment behavior to be used if the thing is unique and has already been added to the container. An ability like "Fly" is likely to be unique, so it's possible that the ability is bootstrapped by multiple sources or even user-selected. When this occurs, you can stipulate the rules for how the new value is assigned to the field. Must be one of these values:
  - **assign:** The new value overrides any existing value.
  - **minimum:** Use the lowest of the new value and any existing value.
  - **maximum:** Use the highest of the new value and any existing value.
  - Default: "assign".

IMPORTANT! There are a variety of important limitations that apply to the assignment of field values via bootstraps. These include the following:

- Non-persistent, derived fields may have their values specified freely.
- All other derived fields and all non-user fields may not be assigned via bootstraps.
- No array-based or matrix-based fields may be assigned via bootstraps.
- User fields may be assigned, but there are additional restrictions (see below).
- Value assignment only occurs for bootstraps that satisfy all their ContainerReq tests.
- If multiple bootstraps attempt to assign the same field value to a unique pick, the values are assigned in the order they are processed by the engine, which is dictated by evaluation timing.

- If two or more value assignments are applied at the exact same timing, then there is *no* guarantee which will take precedence over the other.
- If a bootstrap has ContainerReq tests, the value assignment is applied when the final condition test is resolved for the pick.

Fields of type "user" may be assigned via bootstraps. However, user field incur a number of additional restrictions, as outlined below:

- Cannot be used with conditional bootstraps (only derived fields may)
- Cannot be used with unique things, since the assigned value could overwrite an existing user-modified value
- Specified value is initialized as the default user value when the pick is first added and is thereafter allowed to be modified by the user
- The "maximum" and "minimum" behaviors are pointless with user values, since the value will always be the initial value assigned

EXAMPLE

The following example demonstrates what a "bootstrap" element might look like. All default values are assumed for optional attributes.

```
<bootstrap thing="AttrIncr">
  <containerreq phase="Setup" priority="500">
   val:Level.? >= 4
   </container>
  <autotag group="TheGroup" tag="TheTag"/>
  <assignval field="MyField" value="42"/>
  </bootstrap>
```

CONTAINERREQ ELEMENT (DATA)

THE "CONTAINERREQ" ELEMENT

There will be times when you need to establish a condition for a thing where its container must satisfy a set of criteria. The criteria are spelled out via a Container tag expression. The tag expression is applied against the prospective container of the thing. If the container does not satisfy the tag expression, then the thing must be treated as if it simply doesn't exist within the container. Each separate set of requirements is specified through the use of a "containerreq" element. The complete list of attributes for this element is below.

- **phase:** Id. Specifies the unique id of the evaluation phase during which the tag expression is tested.
- **priority:** Integer. Specifies the evaluation priority during which the tag expression is tested.
- **name:** (Optional) Text. Specifies the name assigned to this test for the purpose of establishing timing dependencies. If empty, no timing dependencies may be defined elsewhere that depend upon this test. Default: Empty.
- **PCDATA:** TagExpr. Specifies the code comprising the Container tag expression.

The "containerreq" element also possesses child elements that pertain to the requirements upon the container. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **match:** An optional "match" element may appear as defined by the given link. This element defines the Match Tag Expression. If omitted, all things are assumed to match and the container requirement test is applied to them all. **IMPORTANT!** This element is only applicable when the container requirement is defined within a component. In all other cases, this element may not be specified.
- **before:** Zero or more "before" elements may appear as defined by the given link. This element specifies appropriate timing dependencies possessed by the container requirement.
- **after:** Zero or more "after" elements may appear as defined by the given link. This element specifies appropriate timing dependencies possessed by the container requirement.

THE "MATCH" ELEMENT

The "match" element defines the Match tag expression that determines whether a particular thing is subject to the container requirement. The tag expression is applied against each thing derived from the component, and the requirement is only assigned to things that satisfy the tag expression. The complete list of attributes for this element is below.

- **PCDATA:** TagExpr. Specifies the code comprising the Match tag expression.

EXAMPLE

The following example demonstrates what a "containerreq" element might look like. All default values are assumed for optional attributes.

```
<containerreq phase="Setup" priority="500" name="MyTest>
  <before name="BeforeTest"/>
  <after name="AfterTest"/>
```

```
val:Level.? >= 4
</containerreq>
```

AUTOADD ELEMENT (DATA)

THE "AUTOADD" ELEMENT

There will likely be situations where you want to pre-select the contents of a chooser or pre-populate items into a table for the user. However, you also want the user to able to delete or replace these items. This is one of the various mechanisms for automatically adding picks to actors and it can be utilized via the "autoadd" element. The complete list of attributes for this element is below.

- **thing:** Id. Specifies the unique id of the thing that is to be automatically added to the container.
- **portal:** Id. Specifies the unique id of the portal into which the thing is to be added.

EXAMPLE

The following example demonstrates what an "autoadd" element might look like. All default values are assumed for optional attributes.

```
<autoadd thing="journal" portal="journal"/>
```

USAGEPOOL ELEMENT (DATA)

THE "USAGEPOOL" ELEMENT

Usage pools provide a convenient way to manage resources that have built-in history tracking of all changes. You define each usage pool via the "usagepool" element. The complete list of attributes for this element is below.

- **id:** Id. Specifies the unique id to utilize for this usage pool.
- **name:** Text. Name to be used in reference to this usage pool. Maximum length is 25 characters.
- **abbrev:** Text. Abbreviation to be used for the usage pool. If empty, the name is also used as the abbreviation. Maximum length is 25 characters. Default: Empty.
- **quantity:** Integer. Specifies the starting quantity to be used for the usage pool. Default: "0".
- **maximum:** Integer. Specifies the maximum history size to be tracked for the pool. A value of zero indicates no limit on the history. Default: "0".
- **ishero:** Boolean. Indicates whether the usage pool is intended for use on each hero/actor or with individual picks. Default: "yes".
- **persist:** Boolean. Indicates whether the history information is saved within the portfolio and restored when it is reloaded. If not persistent, only the latest value is preserved. Default: "no".

EXAMPLE

The following example demonstrates what a "usagepool" element might look like. All default values are assumed for optional attributes.

```
<usagepool
  id="TotalXP"
  name="Total XP"
  abbrev="XP"
  ishero="yes"
  persist="yes"/>
```

REFERENCE ELEMENT (DATA)

THE "REFERENCE" ELEMENT

References provide a mechanism that is very similar to tool tips, with the mouse cursor changing to indicate that help is available for the region beneath the mouse. When the user clicks on the reference, the tip is displayed until the user clicks again to discard it. References are designed for use within encoded text and are ideal for annotating keywords and icons used within the data files. You can define a reference via the "reference" element. The complete list of attributes for this element is below.

- **id:** Id. Specifies the unique id to utilize for this reference. **NOTE!** Unlike most other elements, the unique ids assigned to references may only possess a maximum of 9 characters instead of the usual 10.

- **contents:** Text. Specifies the text to be inserted anywhere the reference is utilized. May contain encoded text.

NOTE! The use of references is not verified at compile time. It is only checked when text containing the reference is rendered in some way. If a reference is undefined when accessed, the text "[Undefined reference: name]" is output. In addition, you can view a list of any undefined references by going to the "Debug" menu and selecting "View Undefined String Usage".

EXAMPLE

The following example demonstrates what a "reference" element might look like. All default values are assumed for optional attributes.

```
<reference
  id="BulletPrf"
  contents="Bullet-Proof Armor"/>
```

GROUP ELEMENT (DATA)

THE "GROUP" ELEMENT

One of the cornerstones of describing objects in the Kit is the tags mechanism. All tags belong to named tag groups, with each tag group having a defined set of valid tag values that can be used. Tag groups also have a number of characteristics which dictate how the tags within that group are handled. Each tag group is specified through the use of a "group" element. The complete list of attributes for this element is below.

- **id:** Id. Specifies the unique id of the tag group. This id is used in all references to the tag group.
- **dynamic:** (Optional) Boolean. Indicates whether the tags for this group can be dynamically defined "on-the-fly" by specifying the tag directly on a thing. If a tag group is not dynamic, then all tags for the group must be defined within the group prior to their use within data files. If a tag group is dynamic, new tags can be defined throughout the data files by simply including the new tag id within a "tag" element on a thing. Dynamic tag groups cannot be assigned "explicit" sequencing. Default: "no".
- **sequence:** (Optional) Set. Designates the sorting sequence to be used for the tags within the group. When you specify the tag group within a sort set, this sequence is used. Must be one of these values:
  - **ascii:** Sort using ASCII codes (digits, all upper case, all lower case).
  - **nocase:** Sort ignoring case (digits, upper case A, lower case A, upper B, etc.).
  - **textvalue:** Treat the name of each tag as a numeric value (if no value, treated as zero for sorting). sequence idvalue – Treat the id of each tag as a numeric value (if no value, treated as zero for sorting).
  - **explicit:** Sort on the value of the "order" attribute assigned to each tag.
  - **id:** Sort on the unique id of each tag as if it were text (allows handling of absolutely any situation).
  - Default: "ascii".
- **minvalue:** (Optional) Integer. Specifies the starting value to use for automatically generating value-based tags (see below). Default: "0".
- **maxvalue:** (Optional) Integer. Specifies the starting value to use for automatically generating value-based tags (see below). Default: "0".

NOTE! Tag groups can automatically generate "value" tags for a given integer range via use of the "minvalue" and/or "maxvalue" attributes. Value tags possess a unique id that is simply an integer value, such as "42". Specifying either or both of these attributes as non-zero makes it easy to create a group of tags number X-Y (e.g. 1-42). When value tags are created, the group automatically uses "idvalue" sequencing. You can define explicit tags for the group that are blended with the auto-generated tags, and any author-defined tag takes precedence over whatever would be auto-generated. Lastly, if the group is dynamic, then additional tags can be defined on- the-fly, as needed.

NOTE! Dynamic tags are processed in the order they are encountered when the data files are compiled, and there is no guarantee of the order in which dynamic tags will be processed. If a dynamic tag has already been defined when it is next encountered, the original definition is used. This means that the first definition of a dynamic tag that happens to be encountered during compilation is the one that will be used for all instances of the tag, so you should strive to ensure that all dynamically defined tags have identical specifications.

The "group" element also possesses child elements that define the individual tags of the group. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **value:** Zero or more "value" elements may appear as defined by the given link. This element specifies the tags that exist for the group.
  NOTE! While it is valid to specify zero tags here, every tag group must have at least one tag defined. You can only specify zero tags here if the group is dynamic and at least one tag is defined directly on a thing.

THE "VALUE" ELEMENT

The "value" element defines a tag for the containing tag group. The complete list of attributes for this element is below.

- **id:** Id. Specifies the unique id of the tag within the group. This id is used in all references to the tag.
- **name:** (Optional) Text. Name assigned to the tag. If empty, the unique id is used as the name. Maximum length is 100 characters. Default: Empty.
- **order:** (Optional) Integer. Specifies the explicit order value to be used when sorting this tag against others within the tag group. This attribute only applies when the tag group is assigned the "explicit" sequence.

EXAMPLE

The following example demonstrates what a "group" element might look like. All default values are assumed for optional attributes.

```
<group id="Equipment" dynamic="yes">
  <value id="Hand" name="Requires one or more hands"/>
  <value id="TwoHand" name="Requires two hands"/>
  <value id="AutoEquip" name="Gear is auto-equipped"/>
  <value id="Natural" name="Natural weapon/armor/etc."/>
  <value id="CustomGear" name="Custom Gear"/>
  </group>
```

SOURCE ELEMENT (DATA)

THE "SOURCE" ELEMENT

Authors can use the sources mechanism as the mean for defining configuration settings that the user can toggle on and off via the "Configure Hero" form. When a given source is enabled by the user, a corresponding tag is automatically added to the actor, which can then be tested within the data files. Each individual source can be defined via the "source" element. The complete list of attributes for this element is below.

- **id:** Id. Specifies the unique id to utilize for this source.
- **name:** Text. Name to be displayed to the user for this source. Maximum length is 50 characters.
- **abbrev:** (Optional) Text. Shortened name to be displayed within the summary of enabled sources. If empty, the name is used as the abbreviation. Maximum length is 50 characters. Default: Empty.
- **description:** (Optional) Text. Description of the source and what behavior(s) it governs within the data files. Default: Empty.
- **parent:** (Optional) Id. Specifies the unique id of a different source that is treated as the parent of this source. All sources are displayed in a hierarchy within the "Configure Hero" form. If empty, this source is presented to the user as a top-level selection. Default: Empty.
- **sortorder:** (Optional) Integer. Assigns an explicit sort order to this source, enabling the display sequence of sources to be controlled by the author (see below). All sources are sequenced in increasing sort order. Default: "99999999".
- **selectable:** (Optional) Boolean. Indicates whether the source is selectable by the user. This option allows sources that are solely used to visually group sources in a hierarchy to be made non-selectable. Default: "yes".
- **maxchoices:** (Optional) Integer - Specifies the maximum number of child sources that can be selected by the user at once. This makes it possible to limit the user to select only one option from a list of sources. If zero, there is no limit to the number of sources that can be selected. Default: "0".
- **minchoices:** (Optional) Integer - Specifies the minimum number of child sources that must be selected by the user beneath this source. This makes it possible to require the user to select one or more options from a list of sources. If zero, there is no requirement to select any sources. Default: "0".
- **reportable:** (Optional) Boolean. Indicates whether the source is included in the hierarchical summary report of selected sources. Allows authors to prune the summary report of unnecessary layers so that the informnation presented to users is both clear and reasonably compact. Default: "yes".
- **reportname:** (Optional) Text. Alternate name to be displayed for the source within the hierarchical summary report. Allows for a more compact name to be used for this purpose. If empty, the name of the source is used in the report. Default: Empty.
- **default:** (Optional) Boolean. Indicates whether the initial (default) state of the source for a new actor is selected or non-selected. Default: "no".

NOTE! Sources that are user-selectable may not also possess child sources. If a user-selectable source is designated as the parent of another source, a compilation error will be reported.

NOTE! When either "minchoices" or "maxchoices" is specified as non-zero, child sources may NOT possess child sources of their own.

NOTE! By default, all sources are sorted alphabetically, based on their name. If an explicit sort order is specified, sources are sorted in increasing order. Sources with children are always sorted AFTER sources without children within the displayed list. This ensures that sources without children are grouped together and directly beneath the parent source for intuitive visual grouping. The handling of sources with/without children takes precedence over any explicit ordering that may be given.

EXAMPLE

The following example demonstrates what a "source" element might look like. All default values are assumed for optional attributes.

```
<source
  id="Output"
  name="Output Options"
  selectable="no"
  description="Assortment of options governing character sheet output">
  </source>

<source
  id="Validation"
  name="Include Validation Report on Sheet"
  abbrev="Show Validation Report"
  parent="Output"
  default="yes"
  description="Whether validation report is included at bottom of character sheet">
  </source>
```

RESOURCE ELEMENT (DATA)

THE "RESOURCE" ELEMENT

Everything associated with fonts, colors, bitmaps and borders is managed via resources. Resources can be defined as top-level elements within structural files. However, they can also be defined as child elements within styles. This latter technique will be more commonly used for bitmaps, since those bitmaps can be defined in conjunction with the style that uses the bitmap (e.g. portals used as buttons and icons).

Each visual attribute used within your data files is specified through the use of a "resource" element. The complete list of attributes for this element is below.

- **id:** Id. Specifies the unique id of the resource. This id is used in all references to the resource.
- **isbuiltin:** (Optional) Boolean. Indicates whether the resource is a "built-in" resource provided by HL for easy re-use. Only bitmaps and borders can be built-in resources, since fonts and colors can be freely defined at any time. Default: "no".
- **issystem:** (Optional) Boolean. Indicates whether the resource is intended to replace a "system" resource utilized by HL. When you want to completely change the visual look of your data files and have that new look integrated into HL's own forms, you will need to specify system resources. Only specific resource ids can be replaced as system resources. Default: "no".

The "resource" element also possesses child elements that define the specifics of the resource. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

IMPORTANT! Exactly one of these child elements may be specified for each resource. If multiple are given, a compiler error will be reported.

- **color:** An optional "color" element may appear as defined by the given link. This element specifies the details of a color resource.
- **font:** An optional "font" element may appear as defined by the given link. This element specifies the details of a font resource.
- **bitmap:** An optional "bitmap" element may appear as defined by the given link. This element specifies the details of a bitmap resource.
- **solid:** An optional "solid" element may appear as defined by the given link. This element specifies the details of a solid-color border resource.
- **border:** An optional "border" element may appear as defined by the given link. This element specifies the details of a bitmap-based border resource.

THE "COLOR" ELEMENT

The "color" element defines the facets of a color resource. The complete list of attributes for this element is below.

- **color:** Text. Color value to be used in the format "xxxxxx". The format for the color uses standard HTML color syntax, with each character representing a hexadecimal digit. The first two characters define the Red color value, the next two Green, and the last two Blue. For example, the color "ff0080" specifies a Red value of "ff", a Green value of "00", and a Blue value of "80". NOTE! For additional details on specifying colors via the HTML syntax, please refer to one of the many websites that provide this information, such as http://www.w3schools.com/Html/html_colors.asp.

THE "FONT" ELEMENT

The "font" element defines the facets of a font resource. The complete list of attributes for this element is below.

- **face:** (Optional) Text. Name of the font face to use. This font must exist on every user's computer, or your data files will not successfully load. Consequently, it is generally a good idea to restrict yourself to the fonts that are provided with every copy of

Windows. For example, you might limit yourself to using either "Arial" or "Times New Roman" for guaranteed support of all HL users. Default: "Arial".

- **size:** (Optiona) Integer. Point size of the font to be used. The size is specified in "quarter points", allowing for fractional font sizes to be defined. Each increment of one equates to 0.25 points of font height. For example, to specify a font size of 10, you would use a value of 40 (10 times 4). For a font size of 10.5, use a value of 42. **NOTE!** Smaller point sizes become indistinguishable from each other when rendered on the screen. For example, there is no visible difference between a size of 16 and 17 (4.0 vs. 4.25 points) on the screen.
- **style:** (Optional) Text. Specifies the font styles to be used for the resource. The various styles available are "bold", "italics", and "underline". You can combine multiple styles by placing a '+' between them (e.g. "bold+italics"). If empty, the "normal" version of the font is utilized, with no special styles applied. Default: Empty.
- **rotation:** (Optional) Set. Designates the rotation angle at which the text will be drawn in this font. Note that rotated text is not always supported in all situations. In addition, all default sizing of portals assumes no rotation is employed, so you'll need to perform your own proper sizing when you use rotated text. Must be one of the following:
    - 0 – No rotation is applied.
    - 45 – Text is rotated 45 degrees.
    - 90 – Text is rotated 90 degrees.
    - 135 – Text is rotated 135 degrees.
    - 180 – Text is rotated 180 degrees.
    - 225 – Text is rotated 225 degrees.
    - 270 – Text is rotated 270 degrees.
    - 315 – Text is rotated 315 degrees.
    - Default: "0"

The "bitmap" Element

The "bitmap" element defines the facets of a bitmap resource. The complete list of attributes for this element is below.

- **bitmap:** Text. Name of the file containing the bitmap image to be used. This file must be placed in the same folder where all of the data files for the game system reside.
- **istransparent:** (Optional) Boolean. Indicates whether the bitmap should be treated as having built-in transparency. Enabling transparency allows you to have bitmaps that appear to be non-rectangular in shape (see below). Default: "no"

NOTE! Transparent bitmaps must be created appropriately so that they will behave transparently. Within the bitmap image, the pixel in the top left corner (position 0,0) is always considered to be transparent. All other pixels within the bitmap that possess the same color are also treated as transparent. The result is a bitmap that defines its own transparency.

The "solid" Element

The "solid" element defines the facets of a solid-color border resource. The complete list of attributes for this element is below.

- **color:** Text. Color value to be used in the format "xxxxxx". The format for the color uses standard HTML color syntax, with each character representing a hexadecimal digit. The first two characters define the Red color value, the next two Green, and the last two Blue. For example, the color "ff0080" specifies a Red value of "ff", a Green value of "00", and a Blue value of "80". **NOTE!** For additional details on specifying colors via the HTML syntax, please refer to one of the many websites that provide this information, such as http://www.w3schools.com/Html/html_colors.asp.
- **thickness:** (Optional) Integer. Specifies the thickness of the border in terms of pixels. Default: "1"

The "border" Element

The "border" element defines the facets of a bitmap-based border resource. Bitmap-based borders are comprised of eight bitmaps. There is one bitmap in each of the four corners around the visual element, plus four additional bitmaps that are used as a repeating pattern along each edge. Each bitmap may also be specified with a mask that is used to draw non-rectangular portions of the bitmap appropriately. Masks must always be monochrome bitmaps, since they identify which pixels of the primary bitmap are and are not drawn.

Unlike most elements, this element possesses no attributes. However, it does have eight child elements that specify the different pieces of the border. The complete list of child elements is below.

- **topleft:** Exactly one "topleft" child element must appear, specifying the upper left corner.
- **topright:** Exactly one "topright" child element must appear, specifying the upper right corner.
- **bottomleft:** Exactly one "bottomleft" child element must appear, specifying the lower left corner.
- **bottomright:** Exactly one "bottomright" child element must appear, specifying the lower right corner.
- **left:** Exactly one "left" child element must appear, specifying the left edge.
- **top:** Exactly one "top" child element must appear, specifying the top edge.
- **right:** Exactly one "right" child element must appear, specifying the right edge.
- **bottom:** Exactly one "bottom" child element must appear, specifying the bottom edge.

IMPORTANT! The bitmaps comprising a border must be symmetric. This means that six of the bitmaps must possess the same height and another six must possess the same width. Specifically, all bitmaps used across the top and bottom edges must have the same height, which includes: topleft, top, topright, bottomleft, bottom, and bottomright. In addition, all bitmaps used across the left and right edges must have the same width, which includes: topleft, left, bottomleft, topright, right, and bottomright.

CORNER ELEMENTS

There are four different corners within a border, resulting in four different child elements: "topleft", "topright", "bottomleft", and "bottomright". Each of these elements has the identical set of attributes, so they are all defined here in one place. The complete list of attributes for these elements is below.

- **bitmap:** Text. Name of the file containing the bitmap image to be used. This file must be placed in the same folder where all of the data files for the game system reside.
- **mask:** (Optional) Text. Name of the file containing the monochromatic mask image to be used. This file must have the same dimensions as the bitmap above and must be placed in the same folder where all of the data files for the game system reside. If empty, the bitmap is assumed to be a solid rectangular region with no transparency. Default: Empty.
- **xoffset:** Reserved for future use. Do not specify.
- **yoffset:** Reserved for future use. Do not specify.

EDGE ELEMENTS

There are four different edges within a border, resulting in four different child elements: "left", "top", "bottom", and "right". Each of these elements has the identical set of attributes, so they are all defined here in one place. The complete list of attributes for these elements is below.

- **bitmap:** Text. Name of the file containing the bitmap image to be used. This file must be placed in the same folder where all of the data files for the game system reside.
- **mask:** (Optional) Text. Name of the file containing the monochromatic mask image to be used. This file must have the mask same dimensions as the bitmap above and must be placed in the same folder where all of the data files for the game system reside. If empty, the bitmap is assumed to be a solid rectangular region with no transparency. Default: Empty.

EXAMPLE

The following example demonstrates what various "resource" elements might look like. All default values are assumed for optional attributes.

```
<resource id="color">
  <color color="ff0080"/>
  </resource>

<resource id="font">
  <font face="Arial" size="40" style="bold"/>
  </resource>

<resource id="bitmap">
  <bitmap bitmap="image.bmp"/>
 </resource>

<resource id="solid">
  <solid color="00ff00" thickness="2"/>
  </resource>

<resource id="border">
  <border>
    <topleft bitmap="topleft.bmp"/>
    <topright bitmap="topright.bmp"/>
    <bottomleft bitmap="bottomleft.bmp"/>
    <bottomright bitmap="bottomright.bmp"/>
    <left bitmap="left.bmp"/>
    <top bitmap="top.bmp"/>
    <right bitmap="right.bmp"/>
    <bottom bitmap="bottom.bmp"/>
    </border>
  </resource>
```

STYLE ELEMENT (DATA)

THE "STYLE" ELEMENT

All of the basic visual look and behaviors of portals is encapsulated in a collection of styles . Each distinct category of portal has its own type of style, and you can only associate styles with portals of the corresponding type. Each separate style is specified through the use of a "style" element. The complete list of attributes for this element is below:

- **id:** Id. Specifies the unique id of the style. This id is used in all references to the style.
- **border:** (Optional) Id. Identifies the border to used in conjunction with this style. All portals may have a border drawn around them, and the border is controlled via the style. You can specify the unique id of the border to use or "none" to indicate no border. Default: "none"

The "style" element also possesses child elements that define the specifics of the style. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

IMPORTANT! With the exception of the "resource" element, exactly one of these child elements may be specified for each style. If multiple are given, a compiler error will be reported. The chosen child element dictates the type of style that is being defined. You may include as many "resource" elements as you wish after the single child element that specifies the style.

- **style_label:** An optional "style_label" element may appear as defined by the given link. This element specifies the details of a style for use with label portals.
- **style_image:** An optional "style_image" element may appear as defined by the given link. This element specifies the details of a style for use with image portals.
- **style_edit:** An optional "style_edit" element may appear as defined by the given link. This element specifies the details of a style for use with edit portals.
- **style_checkbox:** An optional "style_checkbox" element may appear as defined by the given link. This element specifies the details of a style for use with checkbox portals.
- **style_menu:** An optional "style_menu" element may appear as defined by the given link. This element specifies the details of a style for use with menu portals.
- **style_action:** An optional "style_action" element may appear as defined by the given link. This element specifies the details of a style for use with action portals.
- **style_incrementer:** An optional "style_incrementer" element may appear as defined by the given link. This element specifies the details of a style for use with incrementer portals.
- **style_chooser:** An optional "style_chooser" element may appear as defined by the given link. This element specifies the details of a style for use with chooser portals.
- **style_region:** An optional "style_region" element may appear as defined by the given link. This element specifies the details of a style for use with region portals.
- **style_table:** An optional "style_table" element may appear as defined by the given link. This element specifies the details of a style for use with table portals.
- **style_separator:** An optional "style_separator" element may appear as defined by the given link. This element specifies the details of a style for use with separator portals.
- **style_special:** An optional "style_special" element may appear as defined by the given link. This element specifies the details of a style for use with special portals.
- **style_output:** An optional "style_output" element may appear as defined by the given link. This element specifies the details of a style for use with output portals.
- **resource:** Zero or more "resource" elements may appear as defined by the given link. This element specifies new resources that are used in conjunction with the style.

COLORS IN STYLES

Many styles allow you to directly specify a color value in the format "xxxxxx". The format for the color uses standard HTML color syntax, with each character representing a hexadecimal digit. The first two characters define the Red color value, the next two Green, and the last two Blue. For example, the color "ff0080" specifies a Red value of "ff", a Green value of "00", and a Blue value of "80".

For additional details on specifying colors via the HTML syntax, please refer to one of the many websites that provide this information, such as http://www.w3schools.com/Html/html_colors.asp.

SCALING OF IMAGES

When encoded text is supported within a category of portal, you will typically be given control over the scaling of images within the corresponding style. When enabled, image scaling is applied to all bitmaps that are inserted into the encoded text that is rendered into the portal. The scaling ratio is based on the difference between the initial font size for the portal and the current font size at which the text is now being rendered. Scaling is valuable for when you want to ensure that the bitmaps remain proportionally sized relative to the font size of the text, which is important when text and bitmaps are interleaved within the encoded text.

THE "STYLE_LABEL" ELEMENT

The "style_label" element defines the facets of a style for label portals. The complete list of attributes for this element is below.

- **font:** Id. Unique id of the font resource to be used for the style.
- **textcolor:** (Optional) Text. Color value to be used for drawing text in the format "xxxxxx" (see above). If omitted, the "textcolorid" attribute must be specified. Default: Empty.
- **textcolorid:** (Optional) Id. Unique id of the color resource to be used for drawing text. If omitted, the "textcolor" attribute must be specified. Default: Empty.
- **background:** (Optional) Id. Unique id of the bitmap resource to use as the background. If omitted, the text is drawn transparently on the existing background region. Default: Empty.
- **alignment:** (Optional) Set. Specifies how the text should be horizontally aligned within the portal width. Must be one of these values:
  - **left:** Text is left-aligned.
  - **center:** Text is centered within the portal.
  - **right:** Text is right-aligned.
  - **Default:** "left".
- **ispattern:** (Optional) Boolean. Indicates whether the background bitmap should be centered within the portal dimensions or tiled to fill the entire portal. Default: "yes".
- **scaleimage:** (Optional) Boolean. Indicates whether bitmaps included within encoded text are scaled (see above). Default: "no".

THE "STYLE_IMAGE" ELEMENT

The "style_image" element defines the facets of a style for image portals. Since image portals simply contain the image and nothing else, there are no special attributes for this element.

THE "STYLE_EDIT" ELEMENT

The "style_edit" element defines the facets of a style for edit portals. The complete list of attributes for this element is below.

- **font:** Id. Unique id of the font resource to be used for the style.
- **textcolor:** (Optional) Text. Color value to be used for drawing text in the format "xxxxxx" (see above). If omitted, the "textcolorid" attribute must be specified. Default: Empty.
- **textcolorid:** (Optional) Id. Unique id of the color resource to be used for drawing text. If omitted, the "textcolor" attribute must be specified. Default: Empty.
- **backcolor:** (Optional) Text. Color value to be used as the background in the format "xxxxxx" (see above). If omitted, the "backcolorid" attribute must be specified. Default: Empty.
- **backcolorid:** (Optional) Id. Unique id of the color resource to be used as the background. If omitted, the "backcolor" attribute must be specified. Default: Empty.
- **alignment:** (Optional) Set. Specifies how the text should be horizontally aligned within the portal width. Must be one of these values:
  - **left:** Text is left-aligned.
  - **center:** Text is centered within the portal.
  - **right:** Text is right-aligned.
  - **Default:** "left".
- **autoselect:** (Optional) Set. Indicates whether to automatically select the entire text contents of the portal when it gains the focus. Must be one of the following:
  - **yes**: Contents are always automatically selected when focus is gained.
  - **no:** Contents are never selected when focus is gained.
  - **default:** Default handling is performed. If the edit portal contains a numeric field, the contents areautomatically selected, but no selection is performed for text-based fields.
  - **Default:** "default".
- **itemborder:** (Optional) Id. Unique id of the border resource to draw around each individual edit cell for an "edit_date" portal. If "none", no border is drawn. Default: "none". **NOTE!** An "edit_date" portal can be assigned either an item border or a border around the entire portal, but never both.
- **septext:** (Optional) Text. Specifies the text to be drawn between each individual edit cell for an "edit_date" portal. Default: "/".
- **sepfont:** (Optional) Id. Unique id of the font resource to be used for drawing the separator text between individual cells for an "edit_date" portal. If empty, the resource specified by the "font" attribute is assumed. Default: Empty.
- **sepcolor:** (Optional) Text. Color value to be used for drawing the separator text in the format "xxxxxx" (see above). If omitted, the "sepcolorid" attribute must be specified. Default: Empty.
- **sepcolorid:** (Optional) Id. Unique id of the color resource to be used for drawing the separator text. If omitted, the "sepcolor" attribute must be specified. Default: Empty.

THE "STYLE_CHECKBOX" ELEMENT

The "style_checkbox" element defines the facets of a style for checkbox portals. The complete list of attributes for this element is below.

- **font:** Id. Unique id of the font resource to be used for the style.
- **textcolor:** (Optional) Text. Color value to be used for drawing text in the format "xxxxxx" (see above). If omitted, the "textcolorid" attribute must be specified. Default: Empty.
- **textcolorid:** (Optional) Id. Unique id of the color resource to be used for drawing text. If omitted, the "textcolor" attribute must be specified. Default: Empty.
- **scaleimage:** (Optional) Boolean. Indicates whether bitmaps included within encoded text are scaled (see above). Default: "no".
- **check:** (Optional) Id. Unique id of the bitmap resource used to indicate the box is checked. If omitted, a default bitmap is used. Default: Empty.
- **checkoff:** (Optional) Id. Unique id of the bitmap resource used to indicate the box is checked when the portal is disabled. If omitted, a default bitmap is used. Default: Empty.
- **uncheck:** (Optional) Id. Unique id of the bitmap resource used to indicate the box is not checked. If omitted, a default bitmap is used. Default: Empty.
- **uncheckoff:** (Optional) Id. Unique id of the bitmap resource used to indicate the box is not checked when the portal is disabled. If omitted, a default bitmap is used. Default: Empty.

THE "STYLE_MENU" ELEMENT

The "style_menu" element defines the facets of a style for menu portals. The complete list of attributes for this element is below.

- **font:** Id. Unique id of the font resource to be used for the style.
- **textcolor:** (Optional) Text. Color value to be used for drawing text in the format "xxxxxx" (see above). If omitted, the "textcolorid" attribute must be specified. Default: Empty.
- **textcolorid:** (Optional) Id. Unique id of the color resource to be used for drawing text. If omitted, the "textcolor" attribute must be specified. Default: Empty.
- **backcolor:** (Optional) Text. Color value to be used as the background in the format "xxxxxx" (see above). If omitted, the "backcolorid" attribute must be specified. Default: Empty.
- **backcolorid:** (Optional) Id. Unique id of the color resource to be used as the background. If omitted, the "backcolor" attribute must be specified. Default: Empty.
- **selecttext:** (Optional) Text. Color value to be used for drawing text of the selected item when the menu is "dropped" in the format "xxxxxx" (see above). If omitted, the "selecttextid" attribute must be specified. Default: Empty.
- **selecttextid:** (Optional) Id. Unique id of the color resource to be used for drawing text of the selected item when the menu is "dropped". If omitted, the "textcolor" attribute must be specified. Default: Empty.
- **selectback:** (Optional) Text. Color value to be used as the background of the selected item when the menu is "dropped" in the format "xxxxxx" (see above). If omitted, the "selectbackid" attribute must be specified. Default: Empty.
- **selectbackid:** (Optional) Id. Unique id of the color resource to be used as the background of the selected item when the menu is "dropped". If omitted, the "backcolor" attribute must be specified. Default: Empty.
- **activetext:** (Optional) Text. Color value to be used for drawing text of the selected item in the format "xxxxxx" (see above). This color is only used within the "non-dropped" region of the menu and allows the color highlighting of invalid menu items without impacting the behavior of the "dropped" menu. If omitted, the "selectbackid" attribute may be specified. If neither color is specified, the "selecttext" color is used. Default: Empty.
- **activetextid:** (Optional) Id. Unique id of the color resource to be used as the background of the selected item. See the "activetext" attribute above for further details. If omitted, the "activetext" attribute may be specified. Default: Empty.
- **droplist:** (Optional) Id. Unique id of the bitmap to be used for the drop arrow on the right when the menu is enabled. If omitted, the system default bitmap is used. Default: Empty.
- **droplistoff:** (Optional) Id. Unique id of the bitmap to be used for the drop arrow on the right when the menu is disabled. If omitted, the system default bitmap is used. Default: Empty.
- **scaleimage:** (Optional) Boolean. Indicates whether bitmaps included within encoded text are scaled (see above). Default: "no".

THE "STYLE_ACTION" ELEMENT

The "style_action" element defines the facets of a style for action portals (typically used as clickable buttons). The complete list of attributes for this element is below.

- **font:** Id. Unique id of the font resource to be used for the style.
- **textcolor:** (Optional) Text. Color value to be used for drawing text in the format "xxxxxx" (see above). If omitted, the "textcolorid" attribute must be specified. Default: Empty.
- **textcolorid:** (Optional) Id. Unique id of the color resource to be used for drawing text. If omitted, the "textcolor" attribute must be specified. Default: Empty.
- **up:** Id. Unique id of the bitmap resource to be used as the "up" state of the action portal.

- **down:** Id. Unique id of the bitmap resource to be used as the "down" state of the action portal.
- **off:** Id. Unique id of the bitmap resource to be used as the "up" state of the action portal when the portal is disabled.
- **xoffset:** (Optional) Integer. Specifies the offset adjustment of any text along the horizontal X axis. The text is centered along the axis and can be shifted left or right via this attribute, with positive values shifting to the right and negative values shifting left. Default: "0".
- **yoffset:** (Optional) Integer. Specifies the offset adjustment of any text along the vertical Y axis. The text is centered yoffset along the axis and can be shifted up or down via this attribute, with positive values shifting downward and negative values shifting upward. Default: "0".

THE "STYLE_INCREMENTER" ELEMENT

The "style_incrementer" element defines the facets of a style for incrementer portals. The complete list of attributes for this element is below.

- **font:** Id. Unique id of the font resource to be used for the style.
- **textcolor:** (Optional) Text. Color value to be used for drawing text in the format "xxxxxx" (see above). If omitted, the "textcolorid" attribute must be specified. Default: Empty.
- **textcolorid:** (Optional) Id. Unique id of the color resource to be used for drawing text. If omitted, the "textcolor" attribute must be specified. Default: Empty.
- **background:** (Optional) Id. Unique id of the bitmap resource to be used as the background. If empty, the portal has no background and is drawn transparently. Default: Empty.
  NOTE! If a background bitmap is specified, the incrementer size is dictated by the dimensions of the bitmap. If not specified, then the dimensions must be specified via the "fullwidth" and "fullheight" attributes.
- **fullwidth:** (Optional) Integer. Specifies the fixed width to utilize for all incrementers assigned this style. If "0", a background must be specified to dictate the dimensions. Default: "0".
- **fullheight:** (Optional) Integer. Specifies the fixed height to utilize for all incrementers assigned this style. If "0", a background must be specified to dictate the dimensions. Default: "0".
- **textleft:** Integer. Specifies the left edge of the region in which text is drawn within the incrementer.
- **texttop:** Integer. Specifies the top edge of the region in which text is drawn within the incrementer.
- **textwidth:** Integer. Specifies the width of the region in which text is drawn within the incrementer.
- **textheight:** Integer. Specifies the height of the region in which text is drawn within the incrementer.
- **plusup:** Id. Unique id of the bitmap resource to be used for the "+" button in its "up" state.
- **plusdown:** Id. Unique id of the bitmap resource to be used for the "+" button in its "down" state.
- **plusoff:** Id. Unique id of the bitmap resource to be used for the "+" button in its "up" state when the incrementer is disabled.
- **plusx:** Integer. Specifies the offset along the horizontal X axis where the "+" button is positioned within the overall incrementer portal.
- **plusy:** Integer. Specifies the offset along the vertical Y axis where the "+" button is positioned within the overall incrementer portal.
- **minusup:** Id. Unique id of the bitmap resource to be used for the "-" button in its "up" state.
- **minusdown:** Id. Unique id of the bitmap resource to be used for the "-" button in its "down" state.
- **minusoff:** Id. Unique id of the bitmap resource to be used for the "-" button in its "up" state when the incrementer is disabled.
- **minusx:** Integer. Specifies the offset along the horizontal X axis where the "-" button is positioned within the overall incrementer portal.
- **minusy:** Integer. Specifies the offset along the vertical Y axis where the "-" button is positioned within the overall incrementer portal.
- **editable:** (Optional) Boolean. Indicates whether the incrementer value can be directly user-edited by clicking within the editable value region. Disabling this can be useful when the value does not correspond to what is displayed, such as when selecting a die type (e.g. d6, d8, d10). Default: "yes".
- **scaleimage:** (Optional) Boolean. Indicates whether bitmaps included within encoded text are scaled (see above). Default: "no".

THE "STYLE_CHOOSER" ELEMENT

The "style_chooser" element defines the facets of a style for chooser portals. The complete list of attributes for this element is below.

- **font:** Id. Unique id of the font resource to be used for the style.
- **textcolor:** (Optional) Text. Color value to be used for drawing text in the format "xxxxxx" (see above). If omitted, the "textcolorid" attribute must be specified. Default: Empty.
- **textcolorid:** (Optional) Id. Unique id of the color resource to be used for drawing text. If omitted, the "textcolor" attribute must be specified. Default: Empty.
- **backcolor:** (Optional) Text. Color value to be used as the background in the format "xxxxxx" (see above). If omitted, the "backcolorid" attribute must be specified. Default: Empty.

- **backcolorid:** (Optional) Id. Unique id of the color resource to be used as the background. If omitted, the "backcolor" attribute must be specified. Default: Empty.
- **scaleimage:** (Optional) Boolean. Indicates whether bitmaps included within encoded text are scaled (see above). Default: "no".

THE "STYLE_REGION" ELEMENT

The "style_region" element defines the facets of a style for region portals. The complete list of attributes for this element is below.

- **background:** (Optional) Id. Unique id of the bitmap resource to be used as the background for the region. If empty, the portal has background no background and is drawn transparently. Default: Empty.

THE "STYLE_TABLE" ELEMENT

The "style_table" element defines the facets of a style for table portals. The complete list of attributes for this element is below.

- **background:** (Optional) Id. Unique id of the bitmap resource to be used as the background for the table. If empty, the portal has no background and is drawn transparently. Default: Empty.
- **itemborder:** (Optional) Id. Unique id of the bitmap resource to be used as a border around each individual item inside the table. If empty, no border is drawn. Default: Empty.
- **showgridhorz:** (Optional) Boolean. Indicates whether horizontal grid lines should be drawn between each item within the table. Default: "no".
- **showgridvert:** (Optional) Boolean. Indicates whether vertical grid lines should be drawn between each item within the table. Default: "no".
- **gridwidth:** (Optional) Integer. Thickness of the grid lines drawn between items within the table (in pixels). The thickness of both horizontal and vertical grid lines is always the same. Default: "1".
- **gridcolor:** (Optional) Text. Color value to be used for drawing grid lines in the format "xxxxxx" (see above). Default: "888888".
- **gridcolorid:** (Optional) Id. Unique id of the color resource to be used for drawing grid lines. If specified, the default value for the "gridcolor" attribute is ignored. Default: Empty.

THE "STYLE_SEPARATOR" ELEMENT

The "style_separator" element defines the facets of a style for separator portals. The complete list of attributes for this element is below.

- **isvertical:** (Optional) Boolean. Indicates whether the separator is oriented vertically or horizontally. Default: "no".
- **start:** Id. Unique id of the bitmap resource to be used at the left/top end of the separator.
- **end:** Id. Unique id of the bitmap resource to be used at the right/bottom end of the separator.
- **center:** Id. Unique id of the bitmap resource to be used in the middle of the separator. This bitmap is tiled as necessary to fill the entire span of the separator.

THE "STYLE_SPECIAL" ELEMENT

The "style_special" element defines the facets of a style for special portals. Due to its nature, there are no attributes for this element.

THE "STYLE_OUTPUT" ELEMENT

The "style_output" element defines the facets of a style for output portals. The complete list of attributes for this element is below.

- **font:** Id. Unique id of the font resource to be used for the style.
- **textcolor:** (Optional) Text. Color value to be used for drawing text in the format "xxxxxx" (see above). If omitted, the "textcolorid" attribute must be specified. Default: Empty.
- **textcolorid:** (Optional) Id. Unique id of the color resource to be used for drawing text. If omitted, the "textcolor" attribute must be specified. Default: Empty.
- **backcolor:** (Optional) Text. Color value to be used as the background in the format "xxxxxx" (see above). If omitted, the "backcolorid" attribute must be specified. Default: Empty.
- **backcolorid:** (Optional) Id. Unique id of the color resource to be used as the background. If omitted, the "backcolor" attribute must be specified. Default: Empty.
- **alignment:** (Optional) Set. Specifies how the text should be horizontally aligned within the portal width (for labels). Must be one of these values:
    - **left:** Text is left-aligned.
    - **center:** Text is centered within the portal.
    - **right:** Text is right-aligned.
    - **Default:** "left".
- **background:** (Optional) Id. Unique id of the bitmap resource to be used as the background for tables. If empty, the portal has no background and is drawn transparently. Default: Empty.

- **itemborder:** (Optional) Id. Unique id of the bitmap resource to be used as a border around each individual item inside a table. If empty, no border is drawn. Default: Empty.
- **showgridhorz:** (Optional) Boolean. Indicates whether horizontal grid lines should be drawn between each item within a table. Default: "no".
- **showgridvert:** (Optional) Boolean. Indicates whether vertical grid lines should be drawn between each item within a table. Default: "no".
- **gridwidth:** (Optional) Integer. Thickness of the grid lines drawn between items within a table (in pixels). The thickness of both horizontal and vertical grid lines is always the same. Default: "1".
- **gridcolor:** (Optional) Text. Color value to be used for drawing grid lines in the format "xxxxxx" (see above). Default: "888888".
- **gridcolorid:** (Optional) Id. Unique id of the color resource to be used for drawing grid lines. If specified, the default value gridcolorid for the "gridcolor" attribute is ignored. Default: Empty.

EXAMPLE

The following example demonstrates what various "style" elements might look like. All default values are assumed for optional attributes.

```
<style id="label">
  <style_label textcolor="f0f0f0" font="fntnormal" alignment="center"/>
  </style>

<style id="edit" border="sunken">
  <style_edit textcolor="d2d2d2" backcolor="000000" font="fntedit" alignment="center"/>
  </style>

<style id="increment">
  <style_incrementer textcolor="f0f0f0" font="fntincrsim"
     textleft="13" texttop="0" textwidth="24" textheight="20"
     fullwidth="50" fullheight="20"
     plusup="incplusup" plusdown="incplusdn" plusoff="incplusof"
     plusx="39" plusy="0"
     minusup="incminusup" minusdown="incminusdn" minusoff="incminusof"
     minusx="0" minusy="0">
    </style_incrementer>
  </style>

<style id="action">
  <style_action textcolor="000088" font="fntactbig"
     up="actbigup" down="actbigdn" off="actbigof">
   </style_action>
  <resource id="actbigup" isbuiltin="yes">
   <bitmap bitmap="button_big_up.bmp"/>
   </resource>
  <resource id="actbigdn" isbuiltin="yes">
   <bitmap bitmap="button_big_down.bmp"/>
   </resource>
  <resource id="actbigof" isbuiltin="yes">
   <bitmap bitmap="button_big_off.bmp"/>
   </resource>
  </style>

<style id="table" border="brdsystem">
  <style_table itemborder="sunken" showgridhorz="yes" gridcolor="808080"/>
  </style>

<style id="checkbox">
  <style_checkbox textcolor="f0f0f0" font="fntcheck"/>
  </style>

<style id="image" border="brdsystem">
  <style_image/>
  </style>

<style id="menu" border="sunken">
  <style_menu font="fntmenu" textcolor="84c8f7" backcolor="2a2c47"
     selecttext="1414f7" selectback="f0f0f0"/>
  </style>

<style id="separator">
  <style_separator isvertical="no"
     start="sephorzsta" center="sephorzmid" end="sephorzend"/>
  <resource id="sephorzsta" isbuiltin="yes">
   <bitmap bitmap="sep_horz_start.bmp"/>
```

```
  </resource>
 <resource id="sephorzmid" isbuiltin="yes">
  <bitmap bitmap="sep_horz_middle.bmp"/>
  </resource>
 <resource id="sephorzend" isbuiltin="yes">
  <bitmap bitmap="sep_horz_end.bmp"/>
  </resource>
 </style>

<style id="region" border="brdsystem">
 <style_region/>
 </style>
```

COMPONENT ELEMENT (DATA)

THE "COMPONENT" ELEMENT

At the heart of every thing are the components that define their shared behaviors . All components define a collection of fields and re-usable behaviors that can be combined into component sets. Each component is specified through the use of a "component" element. The complete list of attributes for this element is below.

- **id:** Id. Specifies the unique id of the component. This id is used in all references to the component.
- **name:** Text. Public name associated with the component. Maximum length of 50 characters.
- **sequence:** (Optional) Set. Designates the default sorting sequence to be used for picks based on this component. If no sort set is specified, or if the sort set does not yield a difference, picks will be sorted based on this criteria. Must be one of these values:
    - **ascii:** Sort alphabetically based on pick name.
    - **nocase:** Ignore case while sorting alphabetically.
    - **id:** Sort on the unique id of each pick as if it were text.
    - **Default:** "nocase".
- **nonuserosort:** (Optional) Set. Designates the special sorting behavior for picks that are not user-ordered when the same table mixes picks that are user-order with those that are non-user-ordered. Must be one of the following: **NOTE!** Any attempt by the user to move non-orderable picks will fail, as will any attempt to move user-ordered picks above or below non-orderable picks.
    - **first:** Picks that are not user-ordered are sorted before picks that are user-sorted.
    - **last:** Picks that are not user-ordered are sorted after picks that are user-sorted.
    - **none:** No differentiation is made between user-ordered picks and those that are not user-ordered.
    - **Default:** "none".
    - **usernamable:** (Optional) Boolean. Indicates whether picks from this component should be user-namable. If a component allows naming, so does any derived component set, unless the component set explicitly says otherwise. Default: "no".
    - **autocompset:** (Optional) Boolean. Indicates whether the Kit should automatically create a new component set with the identical unique id as the component. This new compset will possess exactly one component - this one. Default: "yes".
    - **orderfield:** (Optional) Id. Unique id to be used when automatically defining a special field for the component that will track the details of user-ordering. If empty, then this component does not support user-ordered sequencing of its picks. Default: Empty.
    - **unwind:** (Optional) Id. Unique id of the tag group to utilize for controlling "unwind" logic of picks derived from this unwind component. The unwind mechanism ensures picks are deleted in the reverse order they are added. If omitted, no unwind logic is enabled for the component. Default: Empty.
    - **isgear:** (Optional) Boolean. Indicates whether things derived from this component are considered to be "gear". The gear mechanism automatically provides additional logic to all things designated as gear. Default: "no".
    - **denyboot:** (Optional) Boolean. Indicates whether things derived from this component are denied the ability to be bootstrapped from any source. Default: "no".
    - **panellink:** (Optional) Id. Unique id of an edit panel that is associated with all things derived from this component as the default panel linkage. If a thing also specifies an explicit panel linkage, that will take precedence. If empty, no default panel linkage is established. Default: Empty.
    - **hasshortname:** (Optional) Boolean. Indicates whether all compsets derived from this component should automatically incorporate the "short name" behavior provided by the Kit. Default: "no".
    - **ispublic:** (Optional) Boolean. Indicates whether this component is displayed within the Editor when the user utilizes the "Find Things" mechanism. In general, internal "helper" components should be made non-public so that the user only interacts with components that will be familiar to them. Default: "yes".

The "component" element also possesses child elements that define various facets of the component. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **field:** Zero or more "field" elements may appear as defined by the given link. This element specifies the fields that exist for the component.
- **usage:** Zero or more "usage" elements may appear as defined by the given link. This element specifies the pick-based usage pools associated with all things derived from the component.
- **linkage:** Zero or more "linkage" elements may appear as defined by the given link. This element specifies the pick linkages that can be defined for all things derived from the component.
- **identity:** Zero or more "identity" elements may appear as defined by the given link. This element specifies any identity tag groups that are defined for the component.
- **containerreq:** Zero or more "containerreq" elements may appear as defined by the given link. This element specifies any container requirements for things derived from the component.
- **displace:** An optional "displace" element may appear as defined by the given link. This element specifies any displacement behaviors that the component must perform.
- **shadow:** An optional "shadow" element may appear as defined by the given link. This element specifies any shadowing behaviors that the component must perform.
- **creation:** An optional "creation" element may appear as defined by the given link. This element defines a Creation Script for the component.
- **deletion:** An optional "deletion" element may appear as defined by the given link. This element defines a Deletion Script for the component.
- **xactsetup:** An optional "xactsetup" element may appear as defined by the given link. This element defines a TransactSetup Script for the component.
- **xactbuy:** An optional "xactbuy" element may appear as defined by the given link. This element defines a TransactBuy Script for the component.
- **xactsell:** An optional "xactsell" element may appear as defined by the given link. This element defines a TransactSell Script for the component.
- **tag:** Zero or more "tag" elements may appear as defined by the given link. This element specifies any tags that are automatically assigned to every thing derived from the component.
- **bootstrap:** Zero or more "bootstrap" elements may appear as defined by the given link. This element specifies any things that are automatically bootstrapped to every thing derived from the component.
- **loadfixup:** An optional "loadfixup" element may appear as defined by the given link. This element defines a LoadFixup Script for the component.
- **eval:** Zero or more "eval" elements may appear as defined by the given link. This element specifies any Eval Scripts that must be performed for the component.
- **evalrule:** Zero or more "evalrule" elements may appear as defined by the given link. This element specifies any EvalRule Scripts that must be performed for the component.
- **prereq:** Zero or more "prereq" elements may appear as defined by the given link. This element specifies any pre-requisite tests that are applied to every thing derived from the component.
- **merge:** An optional "merge" element may appear as defined by the given link. This element defines a Merge Script for the component.
- **split:** An optional "split" element may appear as defined by the given link. This element defines a Split Script for the component.

THE "USAGE" ELEMENT

The "usage" element defines a pick-based usage pool to associate with every thing for the component. The complete list of attributes for this element is below.

- **usage:** Id. Unique id of the usage pool to be associated with all picks derived from this component.

THE "LINKAGE" ELEMENT

The "linkage" element defines a linkage to another pick that is associated with every thing for the component. The complete list of attributes for this element is below.

- **linkage:** Id. Specifies the unique id of the linkage being defined for the component.
- **optional:** (Optional) Boolean. Indicates whether this linkage is optional or required for all picks. Default: "no".

THE "IDENTITY" ELEMENT

The "identity" element defines an identity tag group for the component and all derived things. The complete list of attributes for this element is below.

- **group:** Id. Unique id of the tag group that will be used for identity handling of all things derived from this component. **NOTE!** This tag group does not need to exist. If it does not, it is automatically created and setup appropriately by the Kit, but you may not add further dynamic tags to a tag group that is auto-created in this way.

THE "DISPLACE" ELEMENT
The "displace" element defines the displacement behavior for the component and all derived things. The complete list of attributes for this element is below.

- **target:** (Optional) Set. Designates the target into which picks derived from this component will be displaced. Must be one of these values:
  - **hero:** Picks are displaced into the hero/actor, regardless of location in the structural hierarchy.
  - **parent:** Picks are displaced into the immediate container.
  - **Default:** "hero".
- **PCDATA:** TagExpr. Specifies a tag expression that determines whether an individual pick is displaced. This tag expression is applied to the pick itself - not the container. Furthermore, the tag expression does include tags that are automatically added to the pick at creation, such as auto-tags from a table or a bootstrap. The tag expression is only applied once, when the pick is first added to the container, after which the pick is either displaced or not for the rest of its existence.

THE "SHADOW" ELEMENT
The "shadow" element defines the shadowing behavior for the component and all derived things. The complete list of attributes for this element is below.

- **target:** (Optional) Set. Designates the target into which picks derived from this component will be shadowed. Must be one of these values:
  - **hero:** Picks are shadowed into the hero/actor, regardless of location in the structural hierarchy.
  - **parent:** Picks are shadowed into the immediate container.
  - **Default:** "hero".
- **PCDATA:** TagExpr. Specifies a tag expression that determines whether an individual pick is shadowed. This tag expression is applied to the pick itself - not the container. Furthermore, the tag expression does include tags that are automatically added to the pick at creation, such as auto-tags from a table or a bootstrap. The tag expression is only applied once, when the pick is first added to the container, after which the pick is either shadowed or not for the rest of its existence.

THE "CREATION" ELEMENT
The "creation" element defines a Creation Script for the component. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the Creation script.

THE "DELETION" ELEMENT
The "deletion" element defines a Deletion Script for the component. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the Deletion script.

THE "XACTSETUP" ELEMENT
The "xactsetup" element defines a TransactSetup Script for the component. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the TransactSetup script.

THE "XACTBUY" ELEMENT
The "xactbuy" element defines an TransactBuy Script for the component. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the TransactBuy script.

THE "XACTSELL" ELEMENT
The "xactsell" element defines a TransactSell Script for the component. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the TransactSell script.

THE "LOADFIXUP" ELEMENT
The "loadfixup" element defines a LoadFixup Script for the component. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the LoadFixup script.

THE "MERGE" ELEMENT
The "merge" element defines a Merge Script for the component. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the Merge script.

THE "SPLIT" ELEMENT
The "split" element defines a Split Script for the component. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the Split script.

EXAMPLE

The following example demonstrates what a "component" element might look like. All default values are assumed for optional attributes.

```
<component id="WeaponBase" name="Weapon"
      autocompset="no" hasshortname="yes" panellink="armory">
  <field id="wpDamage" name="Damage" type="derived" maxlength="20"/>
  <field id="wpNetAtk" name="Net Attack" type="derived"/>
  <usage usagepool="mypool"/>
  <linkage linkage="attribute" optional="no"/>
  <identity group="Weapon"/>
  <displace target="hero">Helper.Displace</displace>
  <tag group="Equipment" tag="Hand"/>
  <eval index="1" phase="Final" priority="5000"><![CDATA[
   ~insert script code here
   ]]></eval>
  <evalrule value="1" phase="Validate" priority="5000" message="Resource overspent">
   ~insert script code here
   ]]></evalrule>
  <prereq message="Requirement not met">
   <match>Helper.WeapCheck</match>
   <valid><![CDATA[
    ~insert script code here
    ]]></valid>
   </prereq>
  </component>
```

FIELD ELEMENT (DATA)

THE "FIELD" ELEMENT

Most components will possess at least one field. These fields represents values associated with all things derived from the component, such as attack and damage values for weapons, ranges and components for spells, etc. Extensive details on the use of fields can be found in the separate section on Advanced Fields. Each separate field is specified through the use of a "field" element. The complete list of attributes for this element is below.

- **id:** Id. Specifies the unique id to utilize for this field.
- **name:** Text. Specifies the publicly visible name associated with the field. Maximum length is 30 characters.
- **abbrev:** (Optional) Text. Specifies a shortened abbreviation to use for the field name. If empty, the name is used as the abbreviation and truncated as necessary. Maximum length is 10 characters. Default: Empty.
- **maxlength:** (Optional) Integer. Specifies the maximum number of characters that will be managed for this field. If zero, maxlength the field is considered a value-based field. Default: "0".
- **type:** (Optional) Set. Designates the type of access behavior granted by this field. Field values for picks always start out equal to the value assigned to the thing, and they can often be changed for the pick during evaluation. Must be one of these values:
  - **static:** The value assigned to the thing is fixed and can never be changed.
  - **user:** The value is user-configurable via some appropriate method, such as an edit portal for text-based fields.
  - **derived:** The value is calculated during the evaluation cycle.
  - **Default:** "user".
- **style:** (Optional) Set. Designates any special characteristics of the field. Must be one of these values:
  - **normal:** The field is used normally as a simple value or string.
  - **menu:** The field contains an item selected via a menu.
  - **array:** The field encompasses an array of values, with the size specified via the "arrayrows" attribute.
  - **matrix:** The field encompasses a matrix of values, with the size specified via both the "arrayrows" and "maxtrixcolumns" attributes.
  - **Default:** "normal".
- **decimals:** (Optional) Integer. Specifies the number of decimal places to be included when outputting a value-based field as text. Values are always tracked internally as floating point numbers, but this attribute governs how the field is displayed as text. If zero, the field is always output as an integer value. Default: "0".
- **defvalue:** (Optional) Text. Specifies the default value to be assigned for this field on every thing that possesses the field. If the field is value-based, then this attribute is assumed to be value-based as well and is converted to a value for assignment, with the empty string being treated as zero. Default: "".
- **arrayrows:** (Optional) Integer. Specifies the number of rows possessed by a field designated as either an array or arrayrows matrix. Default: "1".
- **matrixcolumns:** (Optional) Integer. Specifies the number of columns possessed by a field designated as a matrix. Default: "1".

- **minvalue:** (Optional) Integer. Specifies the minimum value that the field can assume. If a value lower than this is ever assigned to the field, it is bounded to this value. Default: "-999999999999999."
- **maxvalue:** (Optional) Integer. Specifies the maximum value that the field can assume. If a value greater than this is ever assigned to the field, it is bounded to this value. Default: "999999999999999."
- **maxfinal:** (Optional) Integer. Specifies the maximum number of characters that any finalized value may possess. If zero, no finalization is performed for the field. If non-zero, a Finalize script must be provided. Default: "0". **NOTE!** Finalization is only supported for value-based fields.
- **nevercache:** (Optional) Boolean. Indicates that the results of finalization for this field should never be cached and always regenerated. This attribute applies only to finalization of values on picks. Default: "no".
- **iscachething:** (Optional) Boolean. Indicates whether the finalized results for this field should be cached for things. Finalized values are rarely utilized for things, but they are expensive to re-calculate when used on things, so this attribute allows things to cache their finalized values. Default: "no".
- **persistence:** (Optional) Set. Designates the persistence behavior for this field, which means how the value is managed across evaluation cycles and whether it is saved/loaded to/from portfolios. Persistence is not applicable for "static" fields and is always performed for "user" fields, so this attribute only applies to "derived" fields. Must be one of these values:
    - **none:** The field has no special persistence behavior, being reset at the start of every evaluation cycle and never saved in portfolios.
    - **noreset:** The field is never reset at the start of evaluation cycles and never saved in portfolios.
    - **full:** The field is never reset and is saved in portfolios for restoration when reloaded.
    - Default: "none"
- **usedelta:** (Optional) Boolean. Indicates whether a separate "delta" value should be maintained for the field. Deltas make it possible to easily have users edit values that are intuitive, even though they integrate adjustments that are applied from other effects. The delta is only applicable to "user" fields that are value-based. Default: "no".
- **ismultiline:** (Optional) Boolean. Indicates whether the field should always be considered to contain multi-line text. If a field is multi-line, all portals mapped to the field are implicitly designated as multi-line. If a field is not multi-line, then portals can make their own determination regarding multi-line behavior. This attribute only applies to text-based fields. Default: "no".
- **signed:** (Optional) Boolean. Indicates whether the field should always be considered to contain a signed value. If signed field is signed, then any portal mapped to the field is implicitly designated as signed, else portals are free to make their own determination regarding signed behavior. Default: "no".
- **format:** (Optional) Set. Designates any special numeric formatting to be enforced for a value-based field. Must be one of these values:
    - **integer:** Any portal mapped to the field will always treat the field as an integer value.
    - **float:** Any portal mapped to the field will always treat the field as a floating-point value.
    - **any:** Portals are free to determine how to present the field for editing.
    - Default: "any".
- **history:** (Optional) Set. Designates what type of history tracking is to be performed for this field. Must be one of these values: **NOTE!** History tracking is only supported for fields that meet the following criteria: type must be "derived"; style must be "normal"; must be value-based; may not be persistent; may not possess a Calculate or Bound script.
    - **none:** No history tracking is performed.
    - **stack:** Tracks all changes in the order applied.
    - **changes:** Tracks all changes in the order applied, ignoring any adjustments that yield no actual change (e.g. "+0" or "*1").
    - **best:** Records only the single largest adjustment, and all changes must use the same modify operator. If an adjustment results in no actual change, it is ignored.
    - **Default:** "none".

The "field" element also possesses child elements that pertain to the individual field value. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **bound:** An optional "bound" element may appear as defined by the given link. This element defines the Bound Script that sets up the bounding limits to be used for the field value. If omitted, bounding is performed against the "minvalue" and "maxvalue" attributes.
  NOTE! A Bound script may only be used on a value-based field.
- **calculate:** An optional "calculate" element may appear as defined by the given link. This element defines the Calculate Script that computes the value to be utilized for the field. If omitted, the field value is only modified via eval scripts. **NOTE!** A Calculate script may only be used on a "derived" field.
- **finalize:** An optional "finalize" element may appear as defined by the given link. This element defines the Finalize Script that synthesizes a final value for display to the user. If omitted, no final formatting is performed. NOTE! A Finalize script may only be used on a value-based field.

THE "BOUND" ELEMENT

The "bound" element defines the Bound script that dynamically calculates the bounding limits for the field value. The complete list of attributes for this element is below.

- **phase:** Id. Specifies the unique id of the evaluation phase during which the script is invoked.
- **priority:** Integer. Specifies the evaluation priority during which the script is invoked.
- **name:** (Optional) Text. Specifies the name assigned to this script for the purpose of establishing timing dependencies. If empty, no timing dependencies may be defined elsewhere that depend upon this script. Default: Empty.
- **isprimary:** (Optional) Boolean. Indicates whether this task should be considered the "primary" task when multiple tasks are assigned the same name. Default: "no".
- **PCDATA:** Script. Specifies the code comprising the Bound script.

THE "CALCULATE" ELEMENT

The "calculate" element defines the Calculate script that dynamically determines the field value. The complete list of attributes for this element is below.

- **phase:** Id. Specifies the unique id of the evaluation phase during which the script is invoked.
- **priority:** Integer. Specifies the evaluation priority during which the script is invoked.
- **name:** (Optional) Text. Specifies the name assigned to this script for the purpose of establishing timing dependencies. If empty, no timing dependencies may be defined elsewhere that depend upon this script. Default: Empty.
- **isprimary:** (Optional) Boolean. Indicates whether this task should be considered the "primary" task when multiple tasks are assigned the same name. Default: "no".
- **PCDATA:** Script. Specifies the code comprising the Bound script.

THE "FINALIZE" ELEMENT

The "finalize" element defines the Finalize script that synthesizes the final value for display to the user. The synthesized value may not exceed the maximum length specified by the "maxfinal" attribute for the field. The complete list of attributes for this element is below.

- **phase:** Id. Specifies the unique id of the evaluation phase during which the script is invoked.
- **priority:** Integer. Specifies the evaluation priority during which the script is invoked.
- **name:** (Optional) Text. Specifies the name assigned to this script for the purpose of establishing timing dependencies. If empty, no timing dependencies may be defined elsewhere that depend upon this script. Default: Empty.
- **isprimary:** (Optional) Boolean. Indicates whether this task should be considered the "primary" task when multiple tasks are assigned the same name. Default: "no".
- **PCDATA:** Script. Specifies the code comprising the Bound script.

EXAMPLE

The following example demonstrates what a "field" element might look like. All default values are assumed for optional attributes.

```
<field
  id="wpDamage"
  name="Damage"
  type="derived"
  maxlength="20">
  </field>

<field
  id="perHeight"
  name="Height"
  type="user"
  maxfinal="20"
  defvalue="68">
  <bound phase="Render" priority="10000"><![CDATA[
    @minimum = field[perHtMin].value
    @maximum = field[perHtMax].value
    ]]></bound>
  <finalize><![CDATA[
    ~calculate the height in terms of feet and inches
    var feet as number
    var inches as number
    feet = @value / 12
    feet = round(feet,0,-1)
    inches = @value - (feet * 12)
    ~synthesize appropriate text to display the height properly
    @text = feet & "'"
    if (inches <> 0) then
```

```
    @text = @text & " " & inches & chr(34)
    endif
  ]]></finalize>
</field>
```

CONTAINERREQ ELEMENT (DATA)

THE "CONTAINERREQ" ELEMENT

There will be times when you need to establish a condition for a thing where its container must satisfy a set of criteria. The criteria are spelled out via a Container tag expression. The tag expression is applied against the prospective container of the thing. If the container does not satisfy the tag expression, then the thing must be treated as if it simply doesn't exist within the container. Each separate set of requirements is specified through the use of a "containerreq" element. The complete list of attributes for this element is below.

- **phase:** Id. Specifies the unique id of the evaluation phase during which the tag expression is tested.
- **priority:** Integer. Specifies the evaluation priority during which the tag expression is tested.
- **name:** (Optional) Text. Specifies the name assigned to this test for the purpose of establishing timing dependencies. If empty, no timing dependencies may be defined elsewhere that depend upon this test. Default: Empty.
- **PCDATA:** TagExpr. Specifies the code comprising the Container tag expression.

The "containerreq" element also possesses child elements that pertain to the requirements upon the container. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **match:** An optional "match" element may appear as defined by the given link. This element defines the Match Tag Expression. If omitted, all things are assumed to match and the container requirement test is applied to them all. **IMPORTANT!** This element is only applicable when the container requirement is defined within a component. In all other cases, this element may not be specified.
- **before:** Zero or more "before" elements may appear as defined by the given link. This element specifies appropriate timing dependencies possessed by the container requirement.
- **after:** Zero or more "after" elements may appear as defined by the given link. This element specifies appropriate timing dependencies possessed by the container requirement.

THE "MATCH" ELEMENT

The "match" element defines the Match tag expression that determines whether a particular thing is subject to the container requirement. The tag expression is applied against each thing derived from the component, and the requirement is only assigned to things that satisfy the tag expression. The complete list of attributes for this element is below:

- **PCDATA:** TagExpr. Specifies the code comprising the Match tag expression.

EXAMPLE

The following example demonstrates what a "containerreq" element might look like. All default values are assumed for optional attributes.

```
<containerreq phase="Setup" priority="500" name="MyTest>
  <before name="BeforeTest"/>
  <after name="AfterTest"/>
  val:Level.? >= 4
  </containerreq>
```

TAG ELEMENT (DATA)

THE "TAG" ELEMENT

When you wish to assign a tag to a component or thing, you will utilize a "tag" element. The complete list of attributes for this element is below.

- **group:** Id. Specifies the unique id of the tag group to which the tag being assigned belongs.
- **tag:** Id. Specifies the unique id of the tag that must be assigned.
- **name:** (Optional) Text. Specifies the name to be used for the tag. If empty, the unique id, as a text string, is used as the name tag's name. Maximum length is 100 characters. Default: Empty.
- **abbrev:** (Optional) Text. Specifies the abbreviation to be used for the tag. If empty, the name is used as the abbreviation, subject to any necessary truncation. Maximum length is 100 characters. Default: Empty.

IMPORTANT! If the tag group is dynamic, you can define new tags by simply assigning them to a component or a thing. When that occurs, you should always specify an appropriate name and abbreviation for the tag. The compiler defines new tags as they are encountered, so there is no way to ensure that one use of a new tag occurs before another, except for the basic rules governing the order in which data files with different file extensions are loaded.

EXAMPLE

The following example demonstrates what a "tag" element might look like. All default values are assumed for optional attributes.

```
<tag group="MyGroup" tag="MyTag" name="Tag Name" abbrev="Abbrev"/>
```

EVAL ELEMENT (DATA)

THE "EVAL" ELEMENT

A major facet of data file authoring it defining scripts on components and things for execution during the evaluation cycle. These scripts are referred to as Eval scripts and are invoked on each individual pick within the actor. Each separate script is specified through the use of an "eval" element. The complete list of attributes for this element is below.

NOTE! For more information on many of these attributes, please see Advanced Script Handling.

- **phase:** Id. Specifies the unique id of the evaluation phase during which the script is invoked.
- **priority:** Integer. Specifies the evaluation priority during which the script is invoked.
- **index:** (Optional) Integer. Assigns an arbitrary, but unique, value to this script. This index is used within error messages to identify the script where the problem occurs. If the script is assigned a name (below), no index is necessary. Default: "1".
- **runlimit:** (Optional) Integer. Specifies the maximum number of times this script will be invoked during the evaluation cycle for each container. A value of zero indicates no limit. Default: "0". **NOTE!** The limit is imposed separately within each container, so the script will always be invoked for picks within different containers.
- **iseach:** (Optional) Boolean. Indicates whether the "runlimit" applies individually to each thing or collectively to all things derived from the component. This attribute is only applicable component scripts. Default: "yes".
- **sortas:** (Optional) Id. Specifies the unique id of a different component for which the script will be sorted when establishing the task evaluation sequence. This attribute is only applicable to component scripts. If empty, the script uses the sequencing for the component it is defined within. Default: Empty.
- **name:** (Optional) Text. Specifies the name assigned to this script for the purpose of establishing timing dependencies. If empty, no timing dependencies may be defined elsewhere that depend upon this script. Default: Empty.
- **isprimary:** (Optional) Boolean. Indicates whether this task should be considered the "primary" task when multiple tasks are assigned the same name. Default: "no".
- **PCDATA:** Script. Specifies the code comprising the Eval script.

The "eval" element also possesses child elements that pertain to the handling of the script. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **match:** An optional "match" element may appear as defined by the given link. This element defines a Match Tag Expression that must be satisfied in order for the script to be assigned to each thing. If omitted, the script is applied to all derived things. **IMPORTANT!** This element is only applicable when the condition test is defined within a component. In all other cases, this element may not be specified.
- **before:** Zero or more "before" elements may appear as defined by the given link. This element specifies appropriate timing dependencies possessed by the script.
- **after:** Zero or more "after" elements may appear as defined by the given link. This element specifies appropriate timing dependencies possessed by the script.

THE "MATCH" ELEMENT

The "match" element defines the Match tag expression that determines whether a particular thing is assigned the eval script. The tag expression is applied against each thing derived from the component, and the script is only assigned to things that satisfy the tag expression. The complete list of attributes for this element is below.

- **PCDATA:** TagExpr. Specifies the code comprising the Match tag expression.

EXAMPLE

The following example demonstrates what an "eval" element might look like. All default values are assumed for optional attributes.

```
<eval index="1" phase="Setup" priority="500">
   <before name="BeforeTest"/>
   <after name="AfterTest"/>
   debug "Script Executing"
   </eval>
```

EvalRule Element (Data)

The "evalrule" Element

Throughout your data files, you'll want to verify facets of different picks within actors. This is accomplished by writing rule scripts on components and things for execution during the evaluation cycle. These rule scripts are referred to as EvalRule scripts, or often simply as "eval rules", and are invoked on each individual pick within the actor. Each separate rule is specified through the use of an "evalrule" element. The complete list of attributes for this element is below.

NOTE! For more information on many of these attributes, please see Advanced Script Handling.

- **phase:** Id. Specifies the unique id of the evaluation phase during which the rule is invoked.
- **priority:** Integer. Specifies the evaluation priority during which the rule is invoked.
- **message:** Text. Specifies the error message displayed to the user within the validation report when the rule is not satisfied.
- **summary:** (Optional) Text. Specifies the summary message displayed within the validation summary bar at the bottom of the main window when the rule is not satisfied. If empty, the message is used as the summary. Default: Empty.
- **index:** (Optional) Integer. Assigns an arbitrary, but unique, value to this rule. This index is used within error messages to identify the rule where the problem occurs. If the rule is assigned a name (below), no index is necessary. Default: "1".
- **severity:** (Optional) Set. Identifies the severity level associated with failing the rule. Must be one of these values:
    - **error:** The rule indicates a serious error within the character
    - **warning:** The rule indicates a non-critical warning within the character
    - **Default:** "error".
- **runlimit:** (Optional) Integer. Specifies the maximum number of times this rule will be invoked during the evaluation cycle for each container. A value of zero indicates no limit. Default: "0". **NOTE!** The limit is imposed separately within each container, so the rule will always be invoked for picks within different containers.
- **iseach:** (Optional) Boolean. Indicates whether the "runlimit" applies individually to each thing or collectively to all things derived from the component. This attribute is only applicable component scripts. Default: "yes".
- **reportlimit:** (Optional) Integer. Specifies the maximum number of times the message for this rule will be reported during the evaluation cycle for each container. A value of zero indicates no limit. Default: "0". NOTE! The limit is imposed separately within each container, so failures of the rule will always be reported for picks within different containers.
- **issilent:** (Optional) Boolean. Indicates whether the rule should report an error message to the user if failed. In some cases, you'll simply want to properly highlight picks as invalid within the interface without reporting individual errors. In such cases, you can have the rule test all picks and mark them as invalid, but suppress any actual message. Default: "no".
- **sortas:** (Optional) Id. Specifies the unique id of a different component for which the rule will be sorted when establishing the task evaluation sequence. This attribute is only applicable to component rules. If empty, the rule uses the sequencing for the component it is defined within. Default: Empty.
- **name:** (Optional) Text. Specifies the name assigned to this rule for the purpose of establishing timing dependencies. If empty, no timing dependencies may be defined elsewhere that depend upon this rule. Default: Empty.
- **isprimary:** (Optional) Boolean. Indicates whether this task should be considered the "primary" task when multiple tasks are assigned the same name. Default: "no".
- **PCDATA:** Script. Specifies the code comprising the EvalRule script.

The "evalrule" element also possesses child elements that pertain to the handling of the rule. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **match:** An optional "match" element may appear as defined by the given link. This element defines a Match Tag Expression that must be satisfied in order for the rule to be assigned to each thing. If omitted, the rule is applied to all derived things. **IMPORTANT!** This element is only applicable when the condition test is defined within a component. In all other cases, this element may not be specified.
- **before:** Zero or more "before" elements may appear as defined by the given link. This element specifies appropriate timing dependencies possessed by the rule.
- **after:** Zero or more "after" elements may appear as defined by the given link. This element specifies appropriate timing dependencies possessed by the rule.

THE "MATCH" ELEMENT

The "match" element defines the Match tag expression that determines whether a particular thing is assigned the eval rule. The tag expression is applied against each thing derived from the component, and the rule is only assigned to things that satisfy the tag expression. The complete list of attributes for this element is below.

- **PCDATA:** TagExpr. Specifies the code comprising the Match tag expression.

EXAMPLE

The following example demonstrates what an "evalrule" element might look like. All default values are assumed for optional attributes.

```
<evalrule index="1" phase="Setup" priority="500" message="Rule Failed">
  <before name="BeforeTest"/>
  <after name="AfterTest"/>
  @valid = 1
  </evalrule>
```

PREREQ ELEMENT (DATA)

THE "PREREQ" ELEMENT

You can establish dependencies wherein certain things require specific conditions to be satisfied by their prospective container in order to be added. These conditions are referred to as Pre-Requisites and are always tested against the prospective container for a thing. Each pre-requisite is specified through the use of a "prereq" element. The complete list of attributes for this element is below.

- **message:** Text. Specifies the error message displayed to the user within the validation report when the pre-requisite is not satisfied.
- **iserror:** (Optional) Boolean. Indicates whether the failing the pre-requisite is considered to an error or merely a warning. This only applies if the user chooses to ignore a failed pre-requisite and add the thing to the container anyways. Default: "yes".
- **onlyonce:** (Optional) Boolean. Indicates whether the pre-requisite should only be reported to the user a single time if it fails. This is important for situations where the pre-requisite is assigned to a thing that is added to a container multiple times, such as class levels within the d20 System. This attribute is only applicable to pre-requisites specified directly on individual things. Default: "no".
- **issilent:** (Optional) Boolean. Indicates whether the pre-requisite should report an error message to the user if failed. In issilent some cases, you'll simply want to properly highlight picks as invalid within the interface without reporting individual errors. In such cases, you can have the pre-requisite perform its tests and mark picks as invalid, but suppress any actual message. Default: "no".

The "prereq" element also possesses child elements that pertain to the handling of the requirement. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **match:** An optional "match" element may appear as defined by the given link. This element defines a Match Tag Expression that must be satisfied in order for the pre-requisite to be assigned to each thing. If omitted, the pre-requisite is applied to all derived things.
  IMPORTANT! This element is only applicable when the pre-requisite is defined within a component. In all other cases, this element may not be specified.
- **test:** An optional "test" element may appear as defined by the given link. This element defines a Container Tag Expression that determines whether the pre-requisite is satisfied. If omitted, the pre-requisite is assumed to satisfy this test.
- **validate:** An optional "valid" element may appear as defined by the given link. This element defines a Validate Script that determines whether the pre-requisite is satisfied. If omitted, the pre-requisite is assumed to satisfy this test.

THE "MATCH" ELEMENT

The "match" element defines the Match tag expression that determines whether a particular thing is assigned the pre-requisite. The tag expression is applied against each thing derived from the component, and the pre-requisite is only assigned to things that satisfy the tag expression. The complete list of attributes for this element is below.

- **PCDATA:** TagExpr. Specifies the code comprising the Match tag expression.

THE "TEST" ELEMENT

The "test" element defines the Container tag expression that determines whether the pre-requisite is satisfied. The tag expression is applied against all the tags of the container, and the pre-requisite is considered valid if the tag expression is satisfied. The complete list of attributes for this element is below.

- **PCDATA:** TagExpr. Specifies the code comprising the Container tag expression.

THE "VALIDATE" ELEMENT

The "validate" element defines the Validate script that determines whether the pre-requisite is satisfied. The script is applied against the container, and the pre-requisite is considered valid if the script reports the container as valid. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the Validate script.

EXAMPLE

The following example demonstrates what an "prereq" element might look like. All default values are assumed for optional attributes.

```
<prereq message="Requirement Failed.">
  <valid>
    @valid = 1
    </valid>
  </prereq>
```

COMPONENT SET ELEMENT (DATA)

THE "COMPSET" ELEMENT

Every "thing" you'll define throughout the data files will be based on a specific component set (or compset). Each component set is defined through the use of a "compset" element. The complete list of attributes for this element is below.

- **id:** Id. Specifies the unique id of the compset. This id is used in all references to the compset.
- **usernamable:** (Optional) Set. Designates whether the user is allowed to rename picks that are based on this compset. Must be one of these values:
    - **yes:** The user is always allowed to rename picks derived from this compset.
    - **no:** The user is never allowed to rename picks derived from this compset.
    - **default:** Whether the user can rename picks derived from this compset is dictated by the components upon which the compset is based. If any component allows renaming, so does the compset.
    - **Default:** "default".
- **stackable:** (Optional) Boolean. Indicates whether picks derived from this compset enable stacking behavior . Default: "no".
- **forceunique:** (Optional) Set. Specifies whether all things derived from this compset must be designated as unique or non-unique. Must be one of these values:
    - **yes:** All things derived from this compset must be designated as unique.
    - **no:** No things derived from this compset may be designated as unique.
    - **default:** Author is free to designate a mixture of unique and non-unique things based on this compset.
    - **Default:** "default".

NOTE! If uniqueness is forced either direction, the compiler will report an error for any thing that is designated incorrectly.

The "compset" element also possesses child elements that pertain to the handling of its components. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **compref:** One or more "compref" elements must appear as defined by the given link. This element specifies the individual components that comprise the compset.
- **distinct:** Zero or more "distinct" elements may appear as defined by the given link. This element identifies additional criteria for determining whether two things can be stacked.

THE "COMPREF" ELEMENT

The "compref" element identifies each of the components that the compset will be built upon. The complete list of attributes for this element is below.

- **component:** Id. Specifies the unique id of the component to be included in the compset.

THE "DISTINCT" ELEMENT

The "distinct" element identifies specially treated fields within the compset (i.e. its components). These fields are used to uniquely identify picks and potentially preclude whether those picks can be stacked with other picks. If two picks have the same values for the specified fields, their stackable behavior is dictated by the normal rules for stackability. However, if any "distinct" field differs, the two picks are considered to be not stackable. The complete list of attributes for this element is below.

NOTE! If no "distinct" fields are specified, normal stacking rules apply. If one or more "distinct" fields are given, all components within the compset must be stackable. Array-based and matrix-based fields cannot be designated as "distinct".

- **field:** Id. Specifies the unique id of the field to be verified as distinct before stacking is allowed.

EXAMPLE

The following example demonstrates what an "compset" element might look like. All default values are assumed for optional attributes.

```
<compset id="Melee" stackable="yes">
  <compref component="WeaponBase"/>
  <compref component="WeapMelee"/>
  <compref component="Equippable"/>
  <compref component="Gear"/>
  </compset>
```

ENTITY ELEMENT (DATA)

THE "ENTITY" ELEMENT

There will be times when you need to create a separate container for picks that is distinct from an actor. For example, a customizable weapon or vehicle can have its own set of picks that tailor that specific object. These objects are referred to as entities and gizmos. Each entity is defined through the use of a "entity" element. The complete list of attributes for this element is below:

- **id:** Id. Specifies the unique id of the entity. This id is used in all references to the entity.
- **form:** (Optional) Id. Specifies the unique id of the form that will be used to edit the contents of all gizmos based on this entity. If empty, there is no ability for the user to directly edit the contents of the gizmo. Default: Empty.
- **defaultthing:** (Optional) Id. Specifies the unique id of a thing to be used as the "default" within the context of the entity. All tables assume a "default" thing of the "actor" pick. However, if you define tables for manipulating picks within a gizmo, attempts to use the "actor" pick will fail. This attribute allows you to specify an alternate pick to be used as the default and must designate a pick that always exists within the gizmo (i.e. a thing that is bootstrapped into it). If empty, no suitable default is setup, but you won't always need one. Default: Empty.

The "entity" element also possesses child elements that pertain to its handling. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **bootstrap:** Zero or more "bootstrap" elements may appear as defined by the given link. This element specifies any things that are automatically bootstrapped into every gizmo derived from the entity.
- **integrity:** An optional "integrity" element may appear as defined by the given link. This element defines an Integrity Script for the entity.

THE "INTEGRITY" ELEMENT

The "integrity" element defines an Integrity Script for the entity that imposes rules for the ensuring the user can only save changes to a valid gizmo. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the Integrity script.

EXAMPLE

The following example demonstrates what an "entity" element might look like. All default values are assumed for optional attributes.

```
<entity id="MyEntity" panel="EntityForm" defaultthing="EntityHelp">
  <bootstrap thing="EntityHelp"/>
  <bootstrap thing="EntityInfo"/>
  </entity>
```

SORT SET ELEMENT (DATA)

THE "SORTSET" ELEMENT

The mechanism used for controlling the sorting sequences of things and picks within the Kit is referred to as "sort sets". Each sort set is defined through the use of a "sortset" element. The complete list of attributes for this element is below:

- **id:** Id. Specifies the unique id of the sort set. This id is used in all references to the sort set.
- **name:** Text. Specifies the public name to be used for the sort set.

The "sortset" element also possesses child elements that pertain to its handling. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **sortkey:** One or more "sortkey" elements must appear as defined by the given link. This element specifies each of the sort criteria to used, and the order in which the sort key are defined dictates the sequence in which they will be applied. Consequently,

the first sort key represents the primary comparison test used; the second sort key is the secondary comparison, etc. The comparisons are applied in order until the objects are differentiated.

THE "SORTKEY" ELEMENT

The "sortkey" element defines the particulars for an individual comparison test that will be employed when sorting. The complete list of attributes for this element is below.

- **id:** Id. Specifies the unique id of the field or tag group to be used for sorting.
- **isfield:** (Optional) Boolean. Indicates whether this sort key will compare objects based on a field value or the presence of tags. It also identifies the nature of the "id" attribute, which will be the unique id of either a field or tag group, as appropriate. Default: "no".
- **isascend:** (Optional) Boolean. Indicates whether the sort key will sequence objects in ascending or descending order. Default: "yes".

EXAMPLE

The following example demonstrates what a "sortset" element might look like. All default values are assumed for optional attributes.

```
<sortset id="Armory" name="Weapons and Armor">
  <sortkey isfield="no" id="Equipped"/>
  <sortkey isfield="no" id="Armory"/>
  <sortkey isfield="no" id="_Name_"/>
  </sortset>
```

DATA FILE REFERENCE

The data file is where you'll be defining all of the top-level elements that the user will directly control. This includes visual elements like panels, layouts, templates, and portals. It also includes game system elements like things.

All data files have the ".dat" file extension and are loaded after all of the structural files have defined the framework for the game system.

This section outlines the structure and mechanics for writing a definition file.

IMPORTANT! This section utilizes critical notational conventions that should be reviewed.

IMPORTANT! Just because you **can** put numerous different elements in the same file does not mean you **should** do so. Keeping your data files small and focused will also keep them much more manageable, so break up all the information across files where appropriate. See the Skeleton data files for examples of this.

STRUCTURAL COMPOSITION

The overall file structure is that of a standard XML file. The file must start with an XML version element in the form: "<?xml version="1.0"?>". Following this, the top-level XML element must be a "document" and it must have a "signature" attribute containing the explicit value "Hero Lab Data".

The following table defines the attributes for a "document" element.

- **signature:** Text. Must be the value "Hero Lab Data".

Within the document element, every data file possesses the following child elements, appearing in the sequence given and with the names specified.

- **procedure:** Zero or more "procedure" elements may appear as defined by the given link. This element specifies a collection of procedures that may be called by scripts throughout the data files.
- **thing:** Zero or more "thing" elements may appear as defined by the given link. This element specifies various thing objects for use within the game system.
- **portal:** Zero or more "portal" elements may appear as defined by the given link. This element specifies an assortment of portals for use within layouts.
- **template:** Zero or more "template" elements may appear as defined by the given link. This element specifies an assortment of templates for use within layouts.
- **layout:** Zero or more "layout" elements may appear as defined by the given link. This element specifies an assortment of layouts for use within panels, forms, and sheets.
- **panel:** Zero or more "panel" elements may appear as defined by the given link. This element specifies a variety of panels that will be presented to the user as part of the game system.

- **form:** Zero or more "form" elements may appear as defined by the given link. This element specifies a variety of forms that will be presented to the user as part of the game system.
- **sheet:** Zero or more "sheet" elements may appear as defined by the given link. This element specifies a variety of sheets that will be used for printouts within dossiers.
- **dossier:** Zero or more "dossier" elements may appear as defined by the given link. This element specifies a collection of dossiers that the user can select for character output.
- **hidden:** Zero or more "hidden" elements may appear as defined by the given link. This element identifies things that HL should remove from use within the game system.
- **editthing:** Zero or more "editthing" elements may appear as defined by the given link. This element specifies a variety of elements that enable use of the integrated Editor for creation of things.
- **faq:** Zero or more "faq" elements may appear as defined by the given link. This element specifies various FAQ entries that will be included within the FAQ page for the game system.

DATA FILE ELEMENTS

PROCEDURE ELEMENT (DATA)

THE "PROCEDURE" ELEMENT

If you need to invoke the same script code from multiple scripts, then you should consider writing a re-usable procedure that defines the code once and can be called from multiple scripts. Each procedure is defined through the use of a "procedure" element. The complete list of attributes for this element is below.

- **id:** Id. Specifies the unique id of the procedure. This id is used in all references to the procedure.
- **context:** (Optional) Set. Designates a general script context from which this procedure is designed to be called. A script context spans multiple different script types that are related in nature and behavior. Must be one of these values:
    - **unknown:** Procedure may only be invoked from within a specific type of script, as dictated by the "scripttype" attribute (below).
    - **pick:** Procedure can be called from any script with a pick as its initial context.
    - **container:** Procedure can be called from any script with a container as its initial context.
    - **entity:** Procedure can be called from entity-related scripts, such as the TitleBar Script.
    - **info:** Procedure can be called from any information synthesis script for a thing or pick, such as the MouseInfo Script.
    - **transact:** Procedure can be called from any transaction script, such as the TransactBuy Script.
    - **combat:** Procedure can be called from any combat-related script, such as the NewCombat Script.
    - **Default:** "unknown".
- **scripttype:** (Optional) Set. Designates the specific script type from which this procedure may be called. The procedure may only be called from scripts of this type. Must be one of these values:
    - **unknown:** Procedure may be invoked from a general category of scripts, as dictated by the "context" attribute (above).
    - **finalize:** Procedure may only be called from within a Finalize Script.
    - **calculate:** Procedure may only be called from within a Calculate Script.
    - **bounds:** Procedure may only be called from within a Bound Script.
    - **eval:** Procedure may only be called from within an Eval Script.
    - **evalrule:** Procedure may only be called from within an EvalRule Script.
    - **mouseinfo:** Procedure may only be called from within a MouseInfo Script.
    - **titlebar:** Procedure may only be called from within a TitleBar Script.
    - **description:** Procedure may only be called from within a Description Script.
    - **trigger:** Procedure may only be called from within a Trigger Script .
    - **label:** Procedure may only be called from within a Label Script.
    - **validate:** Procedure may only be called from within a Validate Script.
    - **xactsetup:** Procedure may only be called from within a TransactSetup Script .
    - **xactbuy:** Procedure may only be called from within a TransactBuy Script.
    - **xactsell:** Procedure may only be called from within a TransactSell Script.
    - **newcombat:** Procedure may only be called from within a NewCombat Script.
    - **endcombat:** Procedure may only be called from within an EndCombat Script.
    - **newturn:** Procedure may only be called from within a NewTurn Script.
    - **integrate:** Procedure may only be called from within an Integrate Script.
    - **synthesize:** Procedure may only be called from within a Synthesize Script.
    - **none:** Procedure may be called from any type of script. However, there is no initial context for identifiers, so no context transitions may be specified that don't explicitly designate a safe initial context (e.g. "hero."). This procedure type is useful when all inputs can be passed via variables and all results returned via variables.
    - **Default:** "unknown".

NOTE! It is valid to setup a focus pick via "setfocus" within a calling script and then utilize the inherited focus pick within a procedure of type "none".

- **PCDATA:** Script. Specifies the code comprising the procedure.

EXAMPLE

The following example demonstrates what a "procedure" element might look like. All default values are assumed for optional attributes.

```
<procedure id="MyProc" context="info">
  ~insert script code here
  </procedure>

<procedure id="MyProc" scripttype="eval">
  ~insert script code here
  </procedure>
```

THING ELEMENT (DATA)

THE "THING" ELEMENT

The vast majority of objects that will comprise your data files are "things", which will be added to actors and customized via scripts. By definition, all behaviors for things are inherited by any derived picks, unless specified otherwise. Each thing is specified through the use of a "thing" element. The complete list of attributes for this element is below.

- **id:** Id. Specifies the unique id of the thing. This id is used in all references to the thing.
- **compset:** Id. Specifies the unique id of the component set from which this thing is derived.
- **name:** Text. Public name associated with the thing. Maximum length of 100 characters.
- **description:** (Optional) Text. Detailed description text associated with the thing, for display to the user. Default: Empty.
- **summary:** (Optional) Text. Brief summary description for the thing, for display to the user in space-constrained situations. If empty, the full description is always used as the summary. Default: Empty.
- **isunique:** (Optional) Boolean. Indicates whether this thing is treated as unique within each container. If unique, a single pick is only ever added, while non-unique things can be separately added any number of times. Default: "no".
- **holdable:** (Optional) Boolean. Indicates whether this thing can be held within other other things, such as backpacks hold various pieces of gear. Default: "yes".
- **maxlimit:** (Optional) Integer. Specifies the maximum number of instances of this thing that can be added to a given container. If zero, there is no limit imposed. Default: "0".
- **panellink:** (Optional) Id. Specifies the unique id of a panel that is officially associated with this thing. Scripts can generically determine the panel linked to a thing to mark it invalid if picks within the panel are invalid. If empty, the default panel linkage defined by components is assumed. Default: Empty.
- **stacking:** (Optional) Set. Designates the default stacking behavior to be assumed whenever this thing is purchased by the user. Must be one of these values:
  - **solo:** Item is created individually, so buying 5 of the thing will add 5 separate instances of the thing to the container.
  - **new:** Item is purchased as a new group, so buying 5 of the thing will add one new instance of the thing stacking that has a quantity of 5.
  - **merge:** Item is merged to any existing instance of the thing within the container, adding the quantity purchased to the existing quantity. If item does not currently exist, "new" behavior is used.
  - **never:** Item can never support stacking.
  - **Default:** "solo".
- **lotsize:** (Optional) Integer. Specifies the number of items typically purchased as a single unit when buying this thing. For example, bullets might be purchased by the box, with a box having 25 bullets in it. Default: "1".
- **replaces:** (Optional) Id. Specifies the unique id of another thing which is completely replaced by this thing. All references to the replaced thing are automatically mapped to this thing, such as bootstrapping. This allows you to wholesale replace a built-in item with new behaviors of your own choosing. If empty, no replacement behavior is utilized. Default: Empty.
- **buytemplate:** (Optional) Id. Specifies the unique id of a template that is used for purchasing the thing. This attribute is only applicable to things which have a child entity and whose purchase cost is variable based on the user-specified composition of the child entity. The specified template is used similarly to the buy template that appears within tables and choosers. However, the template is centered at the bottom of the form used for editing the child gizmo. This mechanism is needed when you have something like a user-customizable magic item within the d20System, where the cost is based on the options selected by the user. If empty, no template is associated with the thing. Default: Empty.
- **xactspecial:** (Optional) Integer. When a buy template is shared between two or more portals or things, the template behavior may need to be tailored based on the usage. If this need arises, this attribute specifies a unique value that identifies this particular

usage. By assigning a different value to each usage and keying on it within the template's Position script, you can tailor the template appropriately. Default: "0".

- **isprivate:** (Optional) Boolean. Indicates whether this thing should be kept private from users within the integrated Editor. When a user creates a new thing as a copy of another thing, all existing things derived from the compset are presented for use as the copy source. By making this thing private, it will not appear for selection. Default: "no".

The "thing" element also possesses child elements that define various facets of the thing. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **fieldval:** Zero or more "fieldval" elements may appear as defined by the given link. This element specifies the starting value for individual fields within the thing.
- **arrayval:** Zero or more "arrayval" elements may appear as defined by the given link. This element specifies the starting value for individual array and matrix elements within the thing.
- **usesource:** Zero or more "usesource" elements may appear as defined by the given link. This element specifies the sources relied upon for this thing to be accessible to the user.
- **tag:** Zero or more "tag" elements may appear as defined by the given link. This element specifies any tags that are automatically assigned to the thing.
- **bootstrap:** Zero or more "bootstrap" elements may appear as defined by the given link. This element specifies any things that are automatically bootstrapped when this thing is added to a container.
- **containerreq:** Zero or more "containerreq" elements may appear as defined by the given link. This element specifies any container requirements for the thing.
- **holdlimit:** An optional "holdlimit" element may appear as defined by the given link. This element defines a HoldLimit Tag Expression that restricts the valid set of gear that can be held by the thing.
- **gear:** An optional "gear" element may appear as defined by the given link. This element defines a Gear Script for the thing to calculate its weight.
- **link:** Zero or more "link" elements may appear as defined by the given link. This element specifies the specific pick linkages that exist for this thing.
- **eval:** Zero or more "eval" elements may appear as defined by the given link. This element specifies any Eval Scripts that must be performed for the thing.
- **evalrule:** Zero or more "evalrule" elements may appear as defined by the given link. This element specifies any EvalRule Scripts that must be performed for the thing.
- **pickreq:** Zero or more "pickreq" elements may appear as defined by the given link. This element specifies any dependencies upon other picks within the container.
- **exprreq:** Zero or more "exprreq" elements may appear as defined by the given link. This element specifies any expression-based dependencies upon the state of the container.
- **prereq:** Zero or more "prereq" elements may appear as defined by the given link. This element specifies any pre-requisite tests that are applied to the thing.
- **child:** An optional "child" element may appear as defined by the given link. This element defines the particulars of a child entity that is attached by the thing.
- **minion:** An optional "minion" element may appear as defined by the given link. This element defines the particulars of a minion that is attached by the thing.

THE "FIELDVAL" ELEMENT

The "fieldval" element defines the starting values to use for various fields within the thing. To initialize elements within arrays and matrices, please see the "arrayval" element (below). The complete list of attributes for this element is below.

- **field:** Id. Unique id of the field for which to specify a starting value.
- **value:** Text. Starting value to assign to the field. If the field is text-based, the value is simply assigned, although it may be truncated if it exceeds the defined maximum length for the field. If the field is value-based, the text is converted to a floating point value and assigned.

THE "ARRAYVAL" ELEMENT

The "arrayval" element defines the starting values to use for individual elements within array and matrix fields of the thing. To initialize elements within fields that are not an array or matrix, please see the "fieldval" attribute (above). The complete list of attributes for this element is below.

- **field:** Id. Unique id of the field for which to specify a starting value. If the field is not an array or matrix, an error is reported.
- **index:** Integer. Specifies the row index of the element to be initialized. The first element of arrays and matrices is index zero.
- **column:** (Optional) Integer. Specifies the column of the element to be initialized within a matrix. The first element of matrices is index zero. This attribute is required for matrix elements and may be omitted for array elements. Default: Empty.

- **value:** Text. Starting value to assign to the field element. If the field is text-based, the value is simply assigned, although it value may be truncated if it exceeds the defined maximum length for the field. If the field is value-based, the text is converted to a floating point value and assigned.

THE "USESOURCE" ELEMENT

The "usesource" element specifies a source that this thing is dependent upon. If the designated source is not enabled by the user, this thing will be treated as not existing for the character. You may define a new source on-the-fly via this element by providing a new unique id and the appropriate additional information. However, you should typically avoid doing so, since you cannot control important facets of sources with just-in-time definition. The complete list of attributes for this element is below.

- **source:** Id. Specifies the unique id of the source with which to establish a dependence.
- **name:** (Optional) Text. Name to be displayed to the user for this source. Maximum length is 50 characters. If the source already exists, this attribute can be left empty. Default: Empty.
- **parent:** (Optional) Id. Specifies the unique id of a different source that is treated as the parent of this source. All sources are displayed in a hierarchy within the "Configure Hero" form. If the source already exists, this attribute can be left empty and the behaviors of the existing source are utilized. If empty and the source does not yet exist, the new source is presented to the user as a top-level selection. Default: Empty.

THE "HOLDLIMIT" ELEMENT

The "holdlimit" element defines a HoldLimit Tag Expression for the thing, which restricts the set of things that can be assigned to this thing as held gear. The tag expression is compared against the tags assigned to each prospective piece of gear that the user wants to move. If the tag expression is satisfied, the gear can be held within the thing, else it cannot. The complete list of attributes for this element is below.

- **PCDATA:** TagExpr. Specifies the code comprising the HoldLimit tag expression.

THE "GEAR" ELEMENT

The "gear" element defines a Gear Script for the thing, which is used to calculate the weight of held items and perform additional gear processing on the thing. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the Gear script.

THE "LINK" ELEMENT

The "link" element defines a linkage to another thing based on the set of possible linkages defined for the underlying components. The complete list of attributes for this element is below.

- **linkage:** Id. Unique id of the linkage that is being setup for this thing.
- **thing:** Id. Unique id of the thing to which the linkage should be established.

THE "PICKREQ" ELEMENT

The "pickreq" element defines a dependency on a specific thing whose existence is important within the container. The complete list of attributes for this element is below.

- **thing:** Id. Unique id of the thing upon which the dependency is being established.
- **iserror:** (Optional) Boolean. Indicates whether failure of the requirement should be treated as an error or merely a warning. Default: "yes".
- **ispreclude:** (Optional) Boolean. Indicates whether the specified thing must exist or must not exist within the container in order for the requirement to be satisfied. Preclusion implies that the thing is not allowed to exist. Default: "no".
- **onlyonce:** (Optional) Boolean. Indicates whether the pre-requisite should only be reported to the user a single time if it fails, regardless of the number of times the thing is added to the container. Default: "no".
- **issilent:** (Optional) Boolean. Indicates whether the pre-requisite should report an error message to the user if failed. Default: "no".

THE "EXPRREQ" ELEMENT

The "exprreq" element defines a dependency on an arithmetic expression that must be satisfied by the container. The complete list of attributes for this element is below.

- **message:** Text. Specifies the message to be reported if the requirement is not satisfied.
- **iserror:** (Optional) Boolean. Indicates whether failure of the requirement should be treated as an error or merely a warning. Default: "yes".
- **onlyonce:** (Optional) Boolean. Indicates whether the pre-requisite should only be reported to the user a single time if it fails, regardless of the number of times the thing is added to the container. Default: "no".

- **issilent:** (Optional) Boolean. Indicates whether the pre-requisite should report an error message to the user if failed. Default: "no".
- **PCDATA:** Text. Specifies an arithmetic expression that is applied to the container to determine whether the requirement is satisfied. The expression must be a valid chunk of script code that can be placed between the parentheses in a standard "if/then" statement (e.g. "if (expr) then").

THE "CHILD" ELEMENT

The "child" element specifies an entity that is always added as a child gizmo of the thing. The complete list of attributes for this element is below.

- **entity:** d. Specifies the unique id of the entity to be attached as a child gizmo.

The "child" element also possesses child elements that tailor the entity. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **tag:** Zero or more "tag" elements may appear as defined by the given link. This element specifies any tags that are automatically assigned to the gizmo when it is created.
- **bootstrap:** Zero or more "bootstrap" elements may appear as defined by the given link. This element specifies any things that are automatically bootstrapped into the gizmo when it is created.

THE "MINION" ELEMENT

The "minion" element specifies that the thing always attaches a minion . The complete list of attributes for this element is below.

- **id:** Id. Specifies the unique id to be used for the minion attached via this thing.
- **ownmode:** (Optional) Boolean. Indicates whether the minion has an independent creation/advancement mode from its master. In some cases, you will want the minion to possess the same behavior as the master and transition with the master. However, if the minion is constructed as an independent character, you may want to handle advancement separately. Default: "yes".
- **isinherit:** (Optional) Boolean. Indicates whether the minion inherits the set of enabled sources from its master. If inheritance is active, the source selections for the master are locked in for the minion, with any changes to the master being immediately reflected within the minion. Default: "no".

The "minion" element also possesses child elements that tailor the minion. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **tag:** Zero or more "tag" elements may appear as defined by the given link. This element specifies any tags that are automatically assigned to the minion when it is created.
- **bootstrap:** Zero or more "bootstrap" elements may appear as defined by the given link. This element specifies any things that are automatically bootstrapped into the minion when it is created.

EXAMPLE

The following example demonstrates what a "thing" element might look like. All default values are assumed for optional attributes.

```
<thing id="MyThing" name="Sample" compset="Race" isunique="yes"
  description="Description goes here">
  <fieldval field="FieldId" value="42"/>
  <tag group="Helper" tag="MyTag"/>
  <bootstrap thing="MyAbility"/>
  <eval phase="Setup" priority="5000">
    ~script code goes here
    </eval>
  </thing>
```

PORTAL ELEMENT (DATA)

THE "PORTAL" ELEMENT

Within the hierarchy of visual elements, the individual elements that the user interacts with are portals. There is a wide assortment of different portal types that can be employed for different purposes. Each separate portal is specified through the use of a "portal" element. The complete list of attributes for this element is below.

- **id:** Id. Specifies the unique id of the portal. This id is used in all references to the portal.
- **style:** Id. Specifies the unique id of the style to utilize with this portal. The style must be compatible with the portal type (e.g. a label style must be used with a label portal).

- **tiptext:** (Optional) Text. Description information to be shown to the user when the user pauses the mouse over the portal. If empty, nothing is shown. Default: Empty.
- **isheader:** (Optional) Boolean. Indicates whether this portal is utilized within a dual-purpose header as part of a table. Default: "no".
  NOTE! This attribute is only permitted on portals within templates that serves as dual-purpose headers. **NOTE!** Field-based portals may never be designated for use within a header, as there is no pick/thing associated with dual-purpose templates.
- **showinvalid:** (Optional) Boolean. Indicates whether the text used within the portal should be automatically changed to the built-in "lblwarning" color when the portal references an invalid pick. This behavior only applies to suitable portal types, including labels, incrementers, checkboxes, and menus. Default: "no".
- **showdisabled:** (Optional) Boolean. Indicates whether the text used within the portal should be automatically changed to the built-in "lbldisable" color when the portal references a pick whose pre-requisites are not satisfied. This behavior only applies to suitable portal types, including labels, incrementers, checkboxes, and menus. Default: "yes".
- **width:** (Optional) Integer. Specifies the default width to use for the portal. In general, portal dimensions should only width be controlled via the Position script of the containing visual element. If zero, the default auto-sizing behaviors are employed. Default: "0".
- **height:** (Optional) Integer. Specifies the default height to use for the portal. In general, portal dimensions should only be controlled via the Position script of the containing visual element. If zero, the default auto-sizing behaviors are employed. Default: "0".

The "portal" element also possesses child elements that define the specifics of the portal. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

IMPORTANT! With the exception of the "live" and "mouseinfo" elements, exactly one of these child elements may be specified for each portal. If multiple are given, a compiler error will be reported. The chosen child element dictates the type of portal that is being defined and its characteristics. You may include up to one "live" and/or "mouseinfo" elements after the single child element that specifies the portal.

- **label:** An optional "label" element may appear as defined by the given link. This element specifies the details of a label portal.
- **image_field:** An optional "image_field" element may appear as defined by the given link. This element specifies the details of a field-based image portal.
- **image_user:** An optional "image_user" element may appear as defined by the given link. This element specifies the details of an image portal containing user-selected images.
- **image_literal:** An optional "image_literal" element may appear as defined by the given link. This element specifies the details of an image portal containing a static image.
- **image_reference:** An optional "image_reference" element may appear as defined by the given link. This element specifies the details of an image portal that references a field-based image.
- **incrementer:** An optional "incrementer" element may appear as defined by the given link. This element specifies the details of an incrementer portal.
- **edit:** An optional "edit" element may appear as defined by the given link. This element specifies the details of an edit portal.
- **edit_date:** An optional "edit_date" element may appear as defined by the given link. This element specifies the details of an editdate portal.
- **checkbox:** An optional "checkbox" element may appear as defined by the given link. This element specifies the details of a checkbox portal.
- **menu_literal:** An optional "menu_literal" element may appear as defined by the given link. This element specifies the details of a menu portal consisting of a fixed set of options.
- **menu_array:** An optional "menu_array" element may appear as defined by the given link. This element specifies the details of an array-based menu portal.
- **menu_things:** An optional "menu_things" element may appear as defined by the given link. This element specifies the details of a thing-based menu portal.
- **action:** An optional "action" element may appear as defined by the given link. This element specifies the details of an action portal.
- **region:** An optional "region" element may appear as defined by the given link. This element specifies the details of a region portal.
- **separator:** An optional "separator" element may appear as defined by the given link. This element specifies the details of a separator portal.
- **chooser_table:** An optional "chooser_table" element may appear as defined by the given link. This element specifies the details of a table-based chooser portal.
- **table_fixed:** An optional "table_fixed" element may appear as defined by the given link. This element specifies the details of a non-editable table portal.

- **table_dynamic:** An optional "table_dynamic" element may appear as defined by the given link. This element specifies the details of a table portal to which the user can add arbitrary items.
- **table_auto:** An optional "table_auto" element may appear as defined by the given link. This element specifies the details of a table portal to which a specific item can be added.
- **setting_edit:** An optional "setting_edit" element may appear as defined by the given link. This element specifies the details of a special portal for editing configuration settings.
- **setting_summary:** An optional "setting_summary" element may appear as defined by the given link. This element specifies the details of a special portal for showing a summary of configuration settings.
- **alliance:** An optional "alliance" element may appear as defined by the given link. This element specifies the details of a special portal for controlling the alliance state of an actor.
- **output_label:** An optional "output_label" element may appear as defined by the given link. This element specifies the details of a label portal for sheet output.
- **output_image:** An optional "output_image" element may appear as defined by the given link. This element specifies the details of an image portal for sheet output.
- **output_table:** An optional "output_table" element may appear as defined by the given link. This element specifies the details of a table portal for sheet output.
- **output_dots:** An optional "output_dots" element may appear as defined by the given link. This element specifies the details of a special label portal for sheet output.
- **output_separator:** An optional "output_separator" element may appear as defined by the given link. This element specifies the details of a separator portal for sheet output.
- **live:** An optional "live" element may appear as defined by the given link. This element defines a Live Tag Expression for the portal.
- **mouseinfo:** An optional "mouseinfo" element may appear as defined by the given link. This element defines a MouseInfo Script for the portal.

The "live" Element

The "live" element defines a Live Tag Expression for the portal that determines whether the portal is applicable based on the prevailing conditions. In general, portals should be controlled via scripts using the "visible" target reference instead of using this mechanism. The complete list of attributes for this element is below.

- **PCDATA:** TagExpr. Specifies the code comprising the Live tag expression.

The "mouseinfo" Element

The "mouseinfo" element defines a MouseInfo Script for the portal that synthesizes text for display to the user whenever the user pauses the mouse over the portal. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the MouseInfo script.

Example

The following example demonstrates what various "portal" elements might look like. All default values are assumed for optional attributes.

```
<portal id="hair" style="editNormal">
  <edit field="perHair"/>
  </portal>

<portal id="name" style="chkNormal" showinvalid="yes"
      tiptext="Click to equip this item">
  <checkbox field="grIsEquip" dynamicfield="grStkName"/>
  </portal>

<portal id="menu" style="menuNormal">
  <menu_things field="adjChosen" component="none" maxvisible="20"
      usepicksfield="adjUsePick" candidatefield="adjCandid">
    <candidate></candidate>
    </menu_things>
  </portal>

<portal id="gender" style="menuNormal">
  <menu_literal field="perGender">
    <choice value="0" display="Gender: Male"/>
    <choice value="1" display="Gender: Female"/>
    </menu_literal>
  </portal>

<portal id="textlist" style="menuNormal">
```

```
  <menu_array field="conTextSel" array="conList" maxvisible="10"/>
  </portal>

<portal id="stRace" style="chsNormal" width="110">
  <chooser_table component="Race" choosetemplate="LargeItem">
    <chosen>
      @text = "Race: " & field[name].text
      </chosen>
    <titlebar>
      @text = "Choose the race for your character"
     </titlebar>
    </chooser_table>
  </portal>
```

LABEL ELEMENT (DATA)

THE "LABEL" ELEMENT

When you want to display text to the user, it's usually easiest to utilize a label portal, which is specified through the use of a "label" element. The complete list of attributes for this element is below.

IMPORTANT! Only one mechanism for specifying the label contents may be employed within a given label portal. That means you may use either the "text" attribute, the "field" attribute, OR the "labeltext" script to define the contents. Use of multiple mechanisms will result in a compilation error.

- **text:** (Optional) Text. Specifies a string of literal text to be displayed within the label. Default: Empty.
- **field:** (Optional) Id. Specifies the unique id of the field whose value is to be displayed within the label. The field must exist within the pick/thing associated with the containing template. If this portal is not defined within a template, a field-based label is not allowed. Default: Empty.
- **ismultiline:** (Optional) Boolean. Indicates whether the label text is to be treated as multi-line or merely a single line of output. Default: "no".
- **scrollable:** (Optional) Boolean. Indicates whether the label text is large enough to require a vertical scroller be included that allows the user to scroll through the contents. Default: "no".
- **istitle:** (Optional) Boolean. Indicates whether the label text is being used as a title, which entails special automatic sizing behaviors for proper handling. Default: "no".

The "label" element also possesses child elements that define additional behaviors of the portal. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **labeltext:** An optional "labeltext" element may appear as defined by the given link. This element defines a Label Script that is used for synthesizing the text to be output.

THE "LABELTEXT" ELEMENT

The "labeltext" element defines a Label Script for the portal. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the Label script.

EXAMPLE

The following example demonstrates what a label portal might look like. All default values are assumed for optional attributes.

```
<portal id="name" style="lblNormal" showinvalid="yes">
  <label field="name"/>
  </portal>

<portal id="cost" style="lblNormal">
  <label>
    <labeltext>
      @text = field[grCost].text
      </labeltext>
    </label>
  </portal>
```

IMAGEFIELD ELEMENT (DATA)

THE "IMAGE_FIELD" ELEMENT

The role of the "image_field" element is to display an image to the user, where the actual image is determined by the contents of a field. If you have a situation where you need to display a different image based on the situation, you have two choices. The first option is to have

separate image portals for each and manage the visibility so only the correct one is shown. The second option is to have one field-based image portal and select the proper image via a script. The latter approach is often easier. The complete list of attributes for this element is below.

- **field:** Id. Specifies the unique id of the field whose contents dictate the image to to be displayed within the portal. The field must be a text-based field and its contents are assumed to specify the filename of a bitmap image within the data file folder for the game system. The field must exist within the pick/thing associated with the containing template. If this portal is not defined within a template, a field-based image is not allowed.
- **istransparent:** (Optional) Boolean. Indicates whether the image should be treated as transparent, wherein the pixel color at istransparent position 0, 0 is considered transparent throughout the image. Default: "no".

EXAMPLE

The following example demonstrates what a field-based image portal might look like. All default values are assumed for optional attributes.

```
<portal id="image" style="imgNormal">
  <image_field field="imagefile"/>
  </portal>
```

IMAGEUSER ELEMENT (DATA)

THE "IMAGE_USER" ELEMENT

The role of the "image_user" element is to allow the user to select the image to be displayed and show it within the portal. Until the user selects an image, a placeholder image is shown instead. This portal type is intended for use in allowing the user to select a character portrait and related situations. The complete list of attributes for this element is below.

- **field:** Id. Specifies the unique id of the field whose contents manage the image to to be displayed within the portal. The field must be a value-based field of "user" style, and its contents should never be modified by script. The field must exist within the pick/thing associated with the containing template. If this portal is not defined within a template, a field-based image is not allowed.
  NOTE! User-selected images are tracked internally with the portfolio, so no link to the original image file exists.

EXAMPLE

The following example demonstrates what a user image portal might look like. All default values are assumed for optional attributes.

```
<portal id="image" style="imgNormal">
  <image_user field="userimage"/>
  </portal>
```

IMAGELITERAL ELEMENT (DATA)

THE "IMAGE_LITERAL" ELEMENT

The role of the "image_literal" element is to display a fixed image to the user, where the image is dictated by the portal definition. Whenever you need to show an icon to the user, such as when indicating a particular condition is present, you can use a literal image portal. The complete list of attributes for this element is below.

- **image:** Text. Specifies the filename of a bitmap image within the data file folder for the game system. The given image is displayed within the portal.
- **istransparent:** (Optional) Boolean. Indicates whether the image should be treated as transparent, wherein the pixel color at position 0, 0 is considered transparent throughout the image. Default: "no".
- **isbuiltin:** (Optional) Boolean. Indicates whether the image file is a "built-in" file provided by HL for easy re-use. Default: "no".

EXAMPLE

The following example demonstrates what a literal image portal might look like. All default values are assumed for optional attributes.

```
<portal id="heldby" style="imgNormal">
  <image_literal image="gearinfo.bmp" istransparent="yes"/>
  <mouseinfo mousepos="middle+above">
    call InfoHeld
    </mouseinfo>
  </portal>
```

IMAGEREFERENCE ELEMENT (DATA)

THE "IMAGE_REFERENCE" ELEMENT

It is not possible to copy a field-based image (including user-added images) to another field. However, there are times when you'll want to do exactly that. Consequently, the Kit provides a mechanism for establishing a reference to an image field. The role of the "image_reference" element is to display the contents of one of these field references. The complete list of attributes for this element is below.

- **field:** Id. Specifies the unique id of the field whose contents dictate the image to to be displayed within the portal. The field must be a value-based field and it must contain a reference to another field that identifies an image. The field must exist within the pick/thing associated with the containing template. If this portal is not defined within a template, a reference-based image is not allowed.

EXAMPLE

The following example demonstrates what a referenace-based image portal might look like. All default values are assumed for optional attributes.

```
<portal id="image" style="imgNormal">
  <image_reference field="imageref"/>
  </portal>
```

INCREMENTER ELEMENT (DATA)

THE "INCREMENTER" ELEMENT

The "incrementer" element comes into play when you want to allow the user to modify a value via "plus" and "minus" adjustments. This approach is ideal for managing attributes, skill ratings, consumption of resources, and a variety of other situations. The complete list of attributes for this element is below.

- **field:** Id. Specifies the unique id of the field whose contents dictate the value displayed within the portal. The field must be a value-based field and the field must exist within the pick/thing associated with the containing template. If this portal is not defined within a template, an incrementer is not allowed.
- **interval:** (Optional) Integer. Specifies the adjustment to be applied whenever the user clicks on the '+' and '-' buttons for the incrementer. Default: "1".

EXAMPLE

The following example demonstrates what an incrementer portal might look like. All default values are assumed for optional attributes.

```
<portal id="adjust" style="incrSimple">
  <incrementer field="adjUser"/>
  </portal>
```

EDIT ELEMENT (DATA)

THE "EDIT" ELEMENT

The "edit" element is used to allow the user to directly enter a value or text that is then saved into a field. This portal is ideal for allowing the user to enter names or descriptions, as well as entering a wide-ranging or arbitrary value. The complete list of attributes for this element is below.

- **field:** Id. Specifies the unique id of the field where the contents to be edited are both retrieved and stored. The field must exist within the pick/thing associated with the containing template. If this portal is not defined within a template, an edit portal is not allowed.
- **maxlength:** (Optional) Integer. Specifies the maximum number of characters that can be entered for this field. If zero, a numeric value will be edited. Default: "0".
- **ismultiline:** (Optional) Boolean. Indicates whether the field should be edited as if it contains multi-line text. This attribute only applies to text-based edit portals. Default: "no".
- **readonly:** (Optional) Boolean. Indicates whether the information within the portal can be edited by the user or is displayed as read-only. Default: "no".
- **format:** (Optional) Set. Designates any special numeric formatting to be enforced for a value-based field. Must be one of these values:
  - **integer:** The contents are edited as an integer value.
  - **float:** The contents are edited as a floating-point value.
  - **any:** No restrictions are imposed on the contents.
  - Default: "any".

- **signed:** (Optional) Boolean. Indicates whether the user can specify a negative value when a value-based field is being edited. Default: "no".

EXAMPLE

The following example demonstrates what an edit portal might look like. All default values are assumed for optional attributes.

```
<portal id="hair" style="editNormal">    <edit field="perHair"/>    </portal>
```

EDITDATE ELEMENT (DATA)

THE "EDIT_DATE" ELEMENT

The "edit_date" element allows the user to edit a structured date or time that adheres to the syntactic rules defined for the game system. If a "game world" date or time is edited, then the proper pieces are presented to the user, as dictated by the structure specified for each within the definition file. If a "real world" date or time is edited, then the traditional pieces are presented. This makes it possible to ensure the user enters a syntactically valid date/time, although semantic rules are not currently enforced. The complete list of attributes for this element is below.

- **field:** Id. Specifies the unique id of the field where the date/time value to be edited is both retrieved and stored. The field must exist within the pick/thing associated with the containing template. If this portal is not defined within a template, an editdate portal is not allowed.
- **readonly:** (Optional) Boolean. Indicates whether the information within the portal can be edited by the user or is displayed as read-only. Default: "no".
- **format:** (Optional) Set. Designates the formatting behavior to be utilized. Must be one of these values:
    - **gamedate:** The contents are edited as a game system date.
    - **gametime:** The contents are edited as a game system time.
    - **realdate:** The contents are edited as a real world date.
    - **realtime:** The contents are edited as a real world time.
    - **Default:** "realdate".

EXAMPLE

The following example demonstrates what an editdate portal might look like. All default values are assumed for optional attributes.

```
<portal id="date" style="editDate">
  <edit_date field="thedate" format="gamedate"/>
  </portal>
```

CHECKBOX ELEMENT (DATA)

THE "CHECKBOX" ELEMENT

The "checkbox" element is useful whenever the user has a choice between two clearly opposite states, such as on/off, enable/disable, show/hide, etc. In addition to the traditional visual presentation of text with a box next to it, the Kit allows you to use checkboxes to present to alternate visual states. For example, within the World of Darkness data files, the abilities to promote items to the top of a list and toggle inclusion within printouts are checkboxes that merely toggle between two states and change the visuals accordingly. The complete list of attributes for this element is below.

- **field:** Id. Specifies the unique id of the field whose contents dictate whether the checkbox is in the on or off state. The field must be a value-based field, with a non-zero value indicating "on" and a zero value indicating "off". The field must exist within the pick/thing associated with the containing template. If this portal is not defined within a template, a checkbox is not allowed.
- **message:** (Optional) Text. Specifies the message text to display next to the actual box. If empty, no text is displayed. Default: Empty.
- **dynamicfield:** (Optional) Id. Specifies the unique id of a different field from which the message text will be pulled. This allows the message text to be dynamically determined via scripts. If empty, no dynamic field is specified. Default: Empty.
- **readonly:** (Optional) Boolean. Indicates whether the checkbox portal is unable to be changed by the user. Default: "no".

EXAMPLE

The following example demonstrates what a checkbox portal might look like. All default values are assumed for optional attributes.

```
<portal id="name" style="chkNormal" showinvalid="yes"
     tiptext="Click to equip this weapon">
  <checkbox field="grIsEquip" dynamicfield="grStkName"/>
  </portal>
```

MENULITERAL ELEMENT (DATA)

THE "MENU_LITERAL" ELEMENT

Menus are useful whenever you need the user to select exactly one item from a collection of options. The role of the "menu_literal" element is to allow you to specify a fixed set of options to choose from. An classic example is the selection of gender on the "Personal" tab, where a menu allows the user to select either male or female. The complete list of attributes for this element is below.

- **field:** Id. Specifies the unique id of the field whose contents reflect the current selection from the menu. The field must be a text-based field and must exist within the pick/thing associated with the containing template. If this portal is not defined within a template, a menu is not allowed.
- **maxvisible:** (Optional) Integer. Specifies the maximum number of items that will be visible at one time within the menu when the user opens it for selection. If there are more items to choose from, a scroller will allow the user to access them. Default: "5".
- **allowuservalue:** (Optional) Boolean. Indicates whether the user is allowed to type in a custom value for use within the menu. If the user does this, the value saved for the menu is simply the text entered by the user. Default: "no".

The "menu_literal" element also possesses child elements that define additional facets of the portal. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **choice:** One or more "choice" elements must appear as defined by the given link. This element defines the individual options that the user can select from.

THE "CHOICE" ELEMENT

The "choice" element defines an option that is listed within the menu for the user to select. Each choice has two facets: the value shown to the user and the value used internally. The sequence in which these elements is defined dictates the sequence in which they will be listed for the user. The complete list of attributes for this element is below.

- **display:** Text. Specifies the text to be displayed for this choice.
- **value:** (Optional) Text. Specifies the text to be tracked internally for this choice. When the user selects the choice, the value is saved into the associated field. If empty, the "display" text is used as the value. If the field used by the menu is value-based, all "value" attributes must be numeric values. Default: Empty.

EXAMPLE

The following example demonstrates what a literal menu portal might look like. All default values are assumed for optional attributes.

```
<portal id="gender" style="menuNormal">
  <menu_literal field="perGender">
   <choice value="0" display="Gender: Male"/>
   <choice value="1" display="Gender: Female"/>
   </menu_literal>
  </portal>
```

MENUARRAY ELEMENT (DATA)

THE "MENU_ARRAY" ELEMENT

The "menu_array" element allows you to create a menu whose options are dynamically determined via scripts. These scripts setup the contents of the array that dictates the available options. The complete list of attributes for this element is below.

- **field:** Id. Specifies the unique id of the field whose contents reflect the current selection from the menu. The field must be a text-based field and must exist within the pick/thing associated with the containing template. If this portal is not defined within a template, a menu is not allowed.
- **array:** Boolean. Specifies the unique id of the array-based field to be used to drive the available options. The field must be text-based, must be an array, and must exist within the pick/thing associated with the containing template.
- **maxvisible:** (Optional) Integer. Specifies the maximum number of items that will be visible at one time within the menu when the user opens it for selection. If there are more items to choose from, a scroller will allow the user to access them. Default: "5".

EXAMPLE

The following example demonstrates what an array-based menu portal might look like. All default values are assumed for optional attributes.

```
<portal id="menutext" style="Menu" width="150">
  <menu_array field="pwmTextSel" array="pwmList" maxvisible="9"/>
  </portal>
```

MENUTHINGS ELEMENT (DATA)

THE "MENU_THINGS" ELEMENT

The "menu_things" element is used when the user needs to select an item that exists within the data files. For example, an in-play adjustment can select an attribute or skill that the actor possesses, or an ability can select a weapon type that receives a bonus. The common thread is that the options presented in the menu are either things or picks within the data files. The complete list of attributes for this element is below.

IMPORTANT! Within a thing-based menu, the selected thing is not added to the container as a new pick. The thing is merely identified for reference and can be accessed via the menu, but no new pick appears within the container. Only tables and choosers can actually add new picks to a container.

- **field:** Id. Specifies the unique id of the field whose contents reflect the current selection from the menu. The field must be a value-based field of style "menu" and must exist within the pick/thing associated with the containing template. If this portal is not defined within a template, a menu is not allowed.
- **component:** Id. Specifies the unique id of the component that all choices must be derived from.
- **defthing:** (Optional) Id. Specifies the unique id of the thing to pre-select as the default choice within the menu. If empty, no default selection is made. Default: Empty.
- **usepicks:** (Optional) Set. Designates whether the menu is populated with things or picks, and, if the latter, where the list of picks is retrieved from. Must be one of these values:
  - **thing:** The menu is populated with things.
  - **container:** The menu is populated with picks from the container of the pick associated with the usepicks template.
  - **hero:** The menu is populated with picks from the actor.
  - **actor:** Same as "hero".
  - **Default:** "thing".
- **sortset:** (Optional) Id. Specifies the unique id of the sort set to be used to determine the sequence in which the items are listed in the menu. If empty, the items are listed alphabetically. Default: Empty.
- **maxvisible:** (Optional) Integer. Specifies the maximum number of items that will be visible at one time within the menu when the user opens it for selection. If there are more items to choose from, a scroller will allow the user to access them. Default: "5".
- **usepicksfield:** (Optional) Id. Specifies the unique id of a value-based field that dynamically dictates when the menu is populated with things or picks. If empty, the "usepicks" attribute dictates the behavior. If specified, the field value dictates the behavior according to the list below. Default: Empty.
  - **0:** The menu is populated with things.
  - **1:** The menu is populated with picks from the container of the pick associated with the template.
  - **2:** The menu is populated with picks from the actor.
- **candidatefield:** (Optional) Id. Specifies the unique id of a text-based field that contains the Candidate tag expression to be used when populating the menu options. Default: Empty. **IMPORTANT!** This mechanism is secondary to the "candidate" element, so the field is ignored if the "candidate" element is non-empty.
- **namefield:** (Optional) Id - Specifies the unique id of a text-based field that is used instead of the "name" of each thing shown within the array. This makes it possible to customize the name to be displayed in the menu differently from the standard name shown for the thing.

The "menu_things" element also possesses child elements that define additional facets of the portal. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **candidate:** An optional "candidate" element may appear as defined by the given link. This element defines a Candidate Tag Expression for the portal.
- **change:** An optional "change" element may appear as defined by the given link. This element defines a Change Script for the portal.

THE "CANDIDATE" ELEMENT

The "candidate" element defines a Candidate Tag Expression for the portal that limits the set of things/picks that are available for selection. The complete list of attributes for this element is below.

- **PCDATA:** TagExpr. Specifies the code comprising the Candidate tag expression.

THE "CHANGE" ELEMENT

The "change" element defines a Change Script for the portal that is invoked whenever the user selects a new choice from the list of options. This script allows the implications of the new selection to be integrated and the display updated. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the Change script.

EXAMPLE

The following example demonstrates what a thing-based menu portal might look like. All default values are assumed for optional attributes.

```
<portal id="menu" style="menuNormal">
  <menu_things field="adjChosen" component="none" maxvisible="20"
       usepicksfield="adjUsePick" candidatefield="adjCandid">
    <candidate></candidate>
    </menu_things>
  </portal>
```

ACTION ELEMENT (DATA)

THE "ACTION" ELEMENT

Action portals behave like buttons and always trigger some sort of behavior. That behavior could be built into HL (e.g. delete a pick) or completely defined by you, but some sort of action is invoked. Action portals are specified via the "action" element. The complete list of attributes for this element is below.

IMPORTANT! Many action portals have behaviors that rely on an associated pick. These portals assume they are defined within the context of a template, and the associated pick is dictated by that template.

- **action:** (Optional) Set. Designates the behavior to be invoked when the action portal is triggered by the user. Must be one of these values:
  - **delete:** Deletes the associated pick. Only useful within tables for allowing the user to delete a pick that they added.
  - **info:** Displays detailed information about the associated pick via the associated MouseInfo script. If no script is defined, default behavior shows the name, failed pre-requisites, and description for the associated pick.
  - **edit:** Brings up a form within which the user can edit the contents of a gizmo, where the gizmo is the child of the associated pick.
  - **form:** Brings up the modal form specified within the "form" attribute.
  - **trigger:** Invokes the Trigger script specified via the "trigger" child element.
  - **gear:** Displays a menu allowing the user to move gear between holders.
  - **notes:** Brings up a form within which the user can edit the contents of the field designated by the "notes" attribute.
  - **load:** Loads the actor containing the associated pick for manipulation by the user.
  - **lock:** Transitions the actor containing the associated pick into advancement mode.
  - **unlock:** Transitions the actor containing the associated pick into creation mode.
  - **master:** Loads the master of the actor containing the associated pick, making it the active actor.
  - **minion:** Loads the minion attached by the associated pick, making it the active actor. If the "minion" attribute specifies a unique id, the designated minion is loaded instead.
  - **getgear:** Displays a menu that allows the user to move gear between actors.
  - **combatstart:** Triggers the start of a new combat within the Tactical Console.
  - **combatend:** Triggers the end of an existing combat within the Tactical Console.
  - **newturn:** Transitions to a new combat turn within the Tactical Console.
  - **initchange:** Incorporates user-made changes to the initiatives of actors within the Tactical Console.
  - **integrate:** Integrates all pending actors into an existing combat within the Tactical Console.
  - **dashsort:** Triggers a re-sort of all actors shown within the Dashboard.
  - **Default:** "delete".
- **buttontext:** (Optional) Text. Specifies the text to draw over whatever bitmap is used for the button. Although buttons with text are often larger that other buttons, they are also much easier to create. If empty, no text is drawn over the bitmap. Default: Empty.
- **confirm:** (Optional) Text. Specifies the text to be displayed as a confirmation message before the triggered behavior is actually performed. This attribute is only utilized for the "trigger", "delete", "edit", and "form" action types. If empty, the triggered behavior is performed immediately - without any confirmation check. Default: Empty.
- **form:** (Optional) Id. Specifies the unique id of the form to be displayed when the portal is triggered. This attribute only applies to the "form" action type and is required for that action type. Default: Empty.
- **field:** (Optional) Id. Specifies the unique id of the field that contains the notes text to be edited. This attribute only applies to the "notes" action type and is required for that action type. Default: Empty.
- **minion:** (Optional) Id. Specifies the unique id of the minion to be switched to when the "minion" action portal is triggered. This attribute only applies to the "minion" action type, but it is not required for that action type. If specified, the indicated minion is accessed. If not specified, the minion attached directly by the pick associated with the portal (via the containing template) is accessed. Default: Empty.

The "action" element also possesses child elements that define additional facets of the portal. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **trigger:** An optional "trigger" element may appear as defined by the given link. This element defines a Trigger Script for the trigger portal.

THE "TRIGGER" ELEMENT

The "trigger" element defines a Trigger Script for the portal that is invoked whenever the user triggers the portal. This script only applies to the "trigger" action type and allows adjustments to be applied to usage pools, such as those for tracking damage, managing journal entries, etc. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the Trigger script.

EXAMPLE

The following example demonstrates what an action portal might look like. All default values are assumed for optional attributes.

```
<portal id="delete" style="actDelete"
      tiptext="Click to delete this equipment">
  <action action="delete"/>
  </portal>

<portal id="addxp" style="actSmall">
  <action action="trigger" buttontext="Add XP">
    <trigger>
      ~add the XP to both the journal entry's and hero's usage pools
      perform usagepool[JrnlXP].adjust[field[jrnXP].value]
      perform hero.usagepool[TotalXP].adjust[field[jrnXP].value]
      perform field[jrnXP].reset
      </trigger>
    </action>
  </portal>
```

REGION ELEMENT (DATA)

THE "REGION" ELEMENT

The role of the "region" element is to allow you to designate a rectangular region that can have a suitable border drawn around it. When you want to put a border around a collection of portals to visually group them, you can use a region element. There are no attributes or child elements for this element.

EXAMPLE

The following example demonstrates what a region portal might look like. All default values are assumed for optional attributes.

```
<portal id="border" style="rgnBorder">
  <region/>
  </portal>
```

SEPARATOR ELEMENT (DATA)

THE "SEPARATOR" ELEMENT

The role of the "separator" element is to insert a vertical or horizontal bar between groupings of portals, acting as a visual separator between them. The complete list of attributes for this element is below.

- **isvertical:** (Optional) Boolean. Indicates whether the separator should be drawn horizontally or vertically. Default: "no".

EXAMPLE

The following example demonstrates what a separator portal might look like. All default values are assumed for optional attributes.

```
<portal id="separator" style="sepHorz">
  <separator isvertical="no"/>
  </portal>
```

CHOOSERTABLE ELEMENT (DATA)

THE "CHOOSER_TABLE" ELEMENT

Choosers are similar to thing-based menus in some ways, as they allow the user to select one thing or pick from a list that is determined dynamically. One key difference with choosers is that any selected thing/pick is added to the container as a new pick. If a pick is selected, a new pick derived from the same thing is added. Another key difference is that the available things/picks are displayed for selection in a "choose form", allowing each object to be presented with detailed information. The "chooser" mechanism is ideal for selecting facets like race, profession, archetype, etc. Each chooser is defined via the use of the "chooser_table" element. The complete list of attributes for this element is below.

- **component:** Id. Specifies the unique id of the component that all selectable objects must be derived from.
- **choosetemplate:** Id. Specifies the unique id of the template to be used for displaying selectable objects.

- **choosepicks:** (Optional) Set. Designates whether the selectable objects consist of things or picks, and, if the latter, where the list of picks is retrieved from. Must be one of these values:
  - **thing:** The selectable objects are things.
  - **container:** The selectable objects are picks from the implicitly identified container. If the containing scene is a form associated with a gizmo, the gizmo is used, else the actor is used.
  - **hero:** The selectable objects are picks from the active actor.
  - **actor:** Same as "hero".
  - **Default:** "thing".
- **choosesortset:** (Optional) Id. Specifies the unique id of the sort set to be used for sequencing all of the objects presented for selection. If empty, all objects are sorted by name. Default: Empty.
- **choosegapx:** (Optional) Integer. Specifies the gap along the X-axis to insert between items presented for selection. Default: "0".
- **choosegapy:** (Optional) Integer. Specifies the gap along the Y-axis to insert between items presented for selection. Default: "0".
- **descwidth:** (Optional) Integer. Specifies the width of the reserved "description" area on the right within the choose form. Some items need more width for lengthy descriptions and some do not, so you can control this as you see fit. Default: "250".
- **buytemplate:** (Optional) Id. Specifies the unique id of the template to be shown in the lower right corner of the choose form for controlling the details of a purchase transaction. If empty, no buy template is utilized. Default: Empty.
- **xactspecial:** (Optional) Integer. When a buy template is shared between two or more portals or things, the template behavior may need to be tailored based on the usage. If this need arises, this attribute specifies a unique value that identifies this particular usage. By assigning a different value to each usage and keying on it within the template's Position script, you can tailor the template appropriately. Default: "0".
- **linkage:** (Optional) Id. Specifies the unique id of a thing that will be used as a linkage. When a new pick is added via the chooser, that pick has an automatic linkage setup to any existing pick derived from the specified thing. If no derived pick exists when the new pick is added, no linkage is ever created. If empty, no linkage is established. Default: Empty.
- **showupdate:** (Optional) Boolean. Indicates whether the chooser needs to be dynamically updated after any modification to the actor so that the influence of other changes are always visually reflected to the user, such as through color highlighting. Default: "no".
  NOTE! This option can significantly slow down display updates on slower computers, so only enable this if truly necessary.
- **candidatepick:** (Optional) Id. Specifies the unique id of a pick that will contain a dynamically generated Candidate tag expression for use in determining the list of available objects to choose from. If empty, the "candidate" child element defines the tag expression to use. Default: Empty.
- **candidatefield:** (Optional) Id. Specifies the unique id of a text-based field that contains the Candidate tag expression used to determine the list of available objects to choose from. This field must exist within the pick identified by the "candidatepick" attribute (above). If empty, the "candidate" child element defines the tag expression to use. Default: Empty.
- **prereqtarget:** (Optional) Set. Designates the container against which all pre-requisite tests need to be performed when determining the list of items available for selection. When displacement is utilized, pre-requisites need to be tested against the container to which the new picks will ultimately be added. Must be one of these values:
  - **container:** The default parent container is used.
  - **parent:** The next parent up the hierarchy is used, which parallels the corresponding displacement target.
  - **hero:** The top-level hero is used, which parallels the corresponding displacement target.
  - **Default:** "container".
- **empty:** (Optional) Text. Specifies the text message to be displayed if the user attempts to select an option and there are no available items to choose from. If empty, a default message is displayed. Default: Empty.

NOTE! Choosers possess a "buy" template, but there is no way to properly "sell" an item selected via a chooser. The reason for this is to allow the "buy" template to be used for customization purposes instead of actually buying and selling gear. The same mechanism can be used to allow the user to configure the item selected via the chooser, such as providing an edit or menu portal to specify an important facet of the selected item.

The "chooser_table" element also possesses child elements that define additional behaviors of the portal. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **candidate:** An optional "candidate" element may appear as defined by the given link. This element defines a Candidate Tag Expression for the portal.
- **needtag:** Zero or more "needtag" elements may appear as defined by the given link. This element defines a tag relationship that must exist between a prospective object and the container in order to list the object among the available items.
- **denytag:** Zero or more "denytag" elements may appear as defined by the given link. This element defines a tag relationship that must not exist between a prospective object and the container in order to list the object among the available items.
- **xacttag:** Zero or more "xacttag" elements may appear as defined by the given link. This element defines a tag that is assigned to the transaction pick while the choose form is visible.
- **secondary:** An optional "secondary" element may appear as defined by the given link. This element defines a Secondary Tag Expression that is associated with every new pick added via the portal.

- **existence:** An optional "existence" element may appear as defined by the given link. This element defines an Existence Tag Expression that is associated with every new pick added via the portal.
- **autotag:** Zero or more "autotag" elements may appear as defined by the given link. This element specifies tags that are automatically assigned to each added thing.
- **chosen:** An optional "chosen" element may appear as defined by the given link. This element defines a Chosen Script for the portal.
- **titlebar:** An optional "titlebar" element may appear as defined by the given link. This element defines a TitleBar Script for the portal.
- **description:** An optional "description" element may appear as defined by the given link. This element defines a Description Script for the portal.
- **change:** An optional "change" element may appear as defined by the given link. This element defines a Change Script for the portal.

THE "CANDIDATE" ELEMENT

The "candidate" element defines a Candidate Tag Expression for the portal that limits the set of things/picks that are available for selection. The complete list of attributes for this element is below.

- **PCDATA:** TagExpr. Specifies the code comprising the Candidate tag expression.

THE "NEEDTAG" ELEMENT

The "needtag" element defines a tag relationship that must exist between the object to be added and the prospective container. Tags from one tag group are enumerated within the container, then the object is tested to make sure that it has at least one matching tag with the same id in a separate tag group. If the tag is not found, the object is not valid for selection and omitted from the available list. The complete list of attributes for this element is below.

- **container:** Id. Specifies the unique id of the tag group to utilize within the container.
- **thing:** Id. Specifies the unique id of the tag group to check within the thing/pick.
- **usehero:** (Optional) Boolean. Indicates whether the container tags are pulled from the prospective container for the new pick or the hero. This distinction can be important when using displacement. Default: "no".

THE "DENYTAG" ELEMENT

The "denytag" element defines a tag relationship that must not exist between the object to be added and the prospective container. Tags from one tag group are enumerated within the container, then the object is tested to make sure that it does not possess any matching tags with the same ids in a separate tag group. If any matching tags are found, the object is not valid for selection and omitted from the available list. The complete list of attributes for this element is below.

- **container:** Id. Specifies the unique id of the tag group to utilize within the container.
- **thing:** Id. Specifies the unique id of the tag group to check within the thing/pick.
- **usehero:** (Optional) Boolean. Indicates whether the container tags are pulled from the prospective container for the new pick or the hero. This distinction can be important when using displacement. Default: "no".

THE "XACTTAG" ELEMENT

The "xacttag" element specifies a tag that is automatically added to the transaction pick while the choose form is shown. These tags allow you to indicate contextual information about where the buy template is being used so that you can tailor the behavior appropriately. The complete list of attributes for this element is below.

- **tag:** Id. Specifies the unique id of the tag to define within the tag group "transact".

THE "SECONDARY" ELEMENT

The "secondary" element defines a Secondary Tag Expression that is automatically associated with every new pick added via the portal. This new tag expression is treated like an additional Container Tag Expression for the pick that must also be satisfied. The complete list of attributes for this element is below.

- **phase:** (Optional) Id. Specifies the unique id of the evaluation phase during which the tag expression is tested. If empty, the default timing is used from the definition file. Default: Empty.
- **priority:** Integer. Specifies the evaluation priority during which the tag expression is tested. If empty, the default timing is used from the definition file. Default: Empty.
- **PCDATA:** TagExpr. Specifies the code comprising the Secondary tag expression.

THE "EXISTENCE" ELEMENT

The "existence" element defines an Existence Tag Expression that is automatically associated with every new pick added via the portal. If a pick ever fails to satisfy the tag expression during an evaluation cycle, the pick is automatically deleted. The complete list of attributes for this element is below.

- **phase:** (Optional) Id. Specifies the unique id of the evaluation phase during which the tag expression is tested. If empty, the default timing is used from the definition file. Default: Empty.
- **priority:** Integer. Specifies the evaluation priority during which the tag expression is tested. If empty, the default timing is used from the definition file. Default: Empty.
- **PCDATA:** TagExpr. Specifies the code comprising the Secondary tag expression.

THE "CHOSEN" ELEMENT

The "chosen" element defines a Chosen Script for the portal, which synthesizes the text to be displayed as the chosen item within the portal. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the Chosen script.

THE "TITLEBAR" ELEMENT

The "titlebar" element defines a TitleBar Script for the portal, which synthesizes the text to be displayed at the top of the choose form. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the TitleBar script.

THE "DESCRIPTION" ELEMENT

The "description" element defines a Description Script for the portal, which synthesizes the text to be displayed within the description region of the choose form for the currently selected item on the left. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the Description script.

THE "CHANGE" ELEMENT

The "change" element defines a Change Script for the portal that is invoked whenever the user selects a new choice from the list of options. This script allows the implications of the new selection to be integrated and the display updated. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the Change script.

EXAMPLE

The following example demonstrates what a choosertable portal might look like. All default values are assumed for optional attributes.

```
<portal id="stRace" style="chsNormal" width="110">
  <chooser_table component="Race" choosetemplate="LargeItem">
    <chosen><![CDATA[
      if (@ispick = 0) then
        @text = "{text ff0000}Select Race"
      else
        @text = "Race: " & field[name].text
        endif
      ]]></chosen>
    <titlebar>
      @text = "Choose the race for your character"
     </titlebar>
    </chooser_table>
  </portal>
```

TABLEFIXED ELEMENT (DATA)

THE "TABLE_FIXED" ELEMENT

Fixed tables present a list of items to the user that cannot be modified by the user through the table. This is ideal for displaying a list of character attributes or a summary of the special abilities possessed by a character. Since they cannot be modified, fixed tables have only a set of behaviors for showing the selected items to the user. Each fixed table is defined via the use of the "table_fixed" element. The complete list of attributes for this element is below.

- **component:** Id. Specifies the unique id of the component that all shown objects must be derived from.
- **showtemplate:** Id. Specifies the unique id of the template to be used for displaying the picks that have been added to the table.
- **showpicks:** (Optional) Set. Designates the source from which the picks shown are retrieved from. Must be one of these values:
  - **container:** The picks shown are from the implicitly identified container. If the containing scene is a form associated with a gizmo, the gizmo is used, else the actor is used.
  - **hero:** The picks shown are from the active actor.
  - **actor:** The picks shown represent all actors in the entire portfolio.
  - **lead:** The picks shown represent all lead actors in the entire portfolio.
  - **minion:** The picks shown are all immediate minions for the active actor.
  - **Default:** "container".

- **showsortset:** (Optional) Id. Specifies the unique id of the sort set to be used for sequencing the items that exist within the table. If empty, all objects are sorted by name. Default: Empty.
- **showgapx:** (Optional) Integer. Specifies the gap along the X-axis to insert between items that exist within the table. Default: "0".
- **showgapy:** (Optional) Integer. Specifies the gap along the Y-axis to insert between items that exist within the table. Default: "0".
- **columns:** (Optional) Integer. Specifies the number of columns of data to display within the table. Default: "1".
- **scrollable:** (Optional) Boolean. Indicates whether the table contents can be scrolled by the user. By default, a scroller is shown whenever the number of items exceeds the visible space, but you can disable this behavior. Default: "yes".
- **headertemplate:** (Optional) Id. Specifies the unique id of the template to be used for a header item that appears at the top headertemplate of the table. This allows you to add column headers above various pieces of information in the table. If empty, the "headertitle" element dictates whether a header is displayed above the table. Default: Empty.
- **headerpick:** (Optional) Id. Specifies the unique id of a thing that is associated with the header item at the top of the table. HL will retrieve any pick based on this thing that exists within the target container and use it. This allows you to control what fields can be accessed from the "headertemplate". If empty, the "actor" pick is used, except within a gizmo, where its "defaultthing" is used instead. Default: Empty.
- **footertemplate:** (Optional) Id. Specifies the unique id of the template to be used for a footer at the bottom of the table. This allows you to put information at the bottom of the table that is always tied to the table for sizing and positioning purposes. If empty, no footer is displayed for the table. Default: Empty.
- **showfixedlast:** (Optional) Boolean. Indicates whether all non-deletable picks within the table are sorted to the end of the list of picks shown. Default: "no".
- **allowuserorder:** (Optional) Boolean. Indicates whether the items in the table can be re-ordered by the user. If enabled, the specified component must designate a suitable ordering field or a separate component with such a field must be specified via the "ordercomponent" attribute. Default: "no". **NOTE!** Verify that whatever sort set you use for showing the items includes the designated ordering field as its first sort key.
- **ordercomponent:** (Optional) Id. Specifies the unique id of an alternate component that possesses a suitable ordering field. This attribute is only applicable when the table supports user ordering. If empty, the ordering field is dictated by the component associated with the table. Default: Empty.
- **alwaysupdate:** (Optional) Boolean. Indicates whether the table must be dynamically updated after any modification to the actor so that the influence of other changes are always visually reflected to the user. Default: "no". **NOTE!** This option can significantly slow down display updates on slower computers, so only enable this if truly necessary.

The "table_fixed" element also possesses child elements that define additional behaviors of the portal. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **list:** An optional "list" element may appear as defined by the given link. This element defines a List Tag Expression for the portal.
- **headertitle:** An optional "headertitle" element may appear as defined by the given link. This element defines a HeaderTitle Script for the portal.

The "list" Element

The "list" element defines a List Tag Expression for the portal that limits the set of picks that are shown. Regardless of this tag expression, all picks added via this portal are always shown within it, enabling deletion of any object added through the table. The complete list of attributes for this element is below.

- **PCDATA:** TagExpr. Specifies the code comprising the List tag expression.

The "headertitle" Element

The "headertitle" element defines a HeaderTitle Script for the portal that synthesizes the text to be displayed at the top of the table as a header. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the HeaderTitle script.

Example

The following example demonstrates what a fixed table portal might look like. All default values are assumed for optional attributes.

```
<portal id="baAttrib" style="tblInvis">
  <table_fixed component="Attribute" scrollable="no"
     showtemplate="baAttrPick" showsortset="explicit">
    <headertitle>
      @text = "Attributes"
      </headertitle>
    </table_fixed>
  </portal>
```

TABLEDYNAMIC ELEMENT (DATA)

THE "TABLE_DYNAMIC" ELEMENT

Dynamic tables allow the user to select the items to be added to the table. As such, they effectively have one set of behaviors for showing the selected items and a separate set of behaviors for selecting the items. A choose form is used to present the list of available items for selection, which allows for detailed information to be shown for each item. When things are added to a table, a new pick is added to the container that is derived from the selected thing. Each dynamic table is defined via the use of the "table_dynamic" element. The complete list of attributes for this element is below.

- **component:** Id. Specifies the unique id of the component that all objects must be derived from, both selectable and shown within the table.
- **showtemplate:** Id. Specifies the unique id of the template to be used for displaying the picks that have been added to the table.
- **showsortset:** (Optional) Id. Specifies the unique id of the sort set to be used for sequencing the items that exist within the table. If empty, all objects are sorted by name. Default: Empty.
- **showgapx:** (Optional) Integer. Specifies the gap along the X-axis to insert between items that exist within the table. Default: "0".
- **showgapy:** (Optional) Integer. Specifies the gap along the Y-axis to insert between items that exist within the table. Default: "0".
- **choosetemplate:** Id. Specifies the unique id of the template to be used for displaying available objects that the user can choose from.
- **choosepicks:** (Optional) Set. Designates whether the selectable objects consist of things or picks, and, if the latter, where the list of picks is retrieved from. Must be one of these values:
  - **thing:** The selectable objects are things.
  - **container:** The selectable objects are picks from the implicitly identified container. If the containing scene is a form associated with a gizmo, the gizmo is used, else the actor is used.
  - **hero:** The selectable objects are picks from the active actor.
  - **Default:** "thing".
- **choosesortset:** (Optional) Id. Specifies the unique id of the sort set to be used for sequencing all of the objects presented for selection. If empty, all objects are sorted by name. Default: Empty.
- **choosegapx:** (Optional) Integer. Specifies the gap along the X-axis to insert between items presented for selection. Default: "0".
- **choosegapy:** (Optional) Integer. Specifies the gap along the Y-axis to insert between items presented for selection. Default: "0".
- **descwidth:** (Optional) Integer. Specifies the width of the reserved "description" area on the right within the choose form. Some items need more width for lengthy descriptions and some do not, so you can control this as you see fit. Default: "250".
- columns: (Optional) Integer. Specifies the number of columns of data to display within the table. Default: "1".
- **scrollable:** (Optional) Boolean. Indicates whether the table contents can be scrolled by the user. By default, a scroller is shown whenever the number of items exceeds the visible space, but you can disable this behavior. Default: "yes".
- **ismultiadd:** (Optional) Boolean. Indicates whether the user can add multiple items at a time to the table, without leaving the choose form. Default: "yes".
- **allowstack:** (Optional) Boolean. Indicates whether the user is allowed to stack items within the table, subject to the restrictions imposed for each item. Default: "yes".
- **addtemplate:** (Optional) Id. Specifies the unique id of the template to be used for the "add" item that always appears at the bottom of a dynamic table and that users will click on to add an item to the table. This allows detailed controlled when the simple "additem" script is not sufficient. If empty, the "additem" element must be specified. Default: Empty.
- **addpick:** (Optional) Id. Specifies the unique id of a thing that is associated with the "add" item at the bottom of the table. HL will retrieve any pick based on this thing that exists within the target container and use it. This allows you to control what fields can be accessed from the "addtemplate". If empty, the "actor" pick is used, except within a gizmo, where its "defaultthing" is used instead. Default: Empty.
- **addspace:** (Optional) Integer. Specifies the additional vertical space to be inserted when displaying the simple "add" item at the bottom of the table by using the "additem" script. The height of the item is based on the font height of the text shown, so this attribute allows you to insert additional padding if you wish. Default: "2".
- **headertemplate:** (Optional) Id. Specifies the unique id of the template to be used for a header item that appears at the top of the table. This allows you to add column headers above various pieces of information in the table. If empty, the "headertitle" element dictates whether a header is displayed above the table. Default: Empty.
- **headerpick:** (Optional) Id. Specifies the unique id of a thing that is associated with the header item at the top of the table. HL will retrieve any pick based on this thing that exists within the target container and use it. This allows you to control what fields can be accessed from the "headertemplate". If empty, the "actor" pick is used, except within a gizmo, where its "defaultthing" is used instead. Default: Empty.
- **buytemplate:** (Optional) Id. Specifies the unique id of the template to be shown in the lower right corner of the choose form for controlling the details of a purchase transaction. If empty, no buy template is utilized. Default: Empty.
- **xactspecial:** (Optional) Integer. When a buy template is shared between two or more portals or things, the template behavior may need to be tailored based on the usage. If this need arises, this attribute specifies a unique value that identifies this particular usage. By assigning a different value to each usage and keying on it within the template's Position script, you can tailor the template appropriately. Default: "0".

- **selltemplate:** (Optional) Id. Specifies the unique id of the template to be shown when the user attempts to delete an item. This allows you to enable the selling of items for money. If empty, no sell template is utilized. Default: Empty.
- **candidatepick:** (Optional) Id. Specifies the unique id of a pick that will contain a dynamically generated Candidate tag expression for use in determining the list of available objects to choose from. If empty, the "candidate" child element defines the tag expression to use. Default: Empty.
- **candidatefield:** (Optional) Id. Specifies the unique id of a text-based field that contains the Candidate tag expression used to determine the list of available objects to choose from. This field must exist within the pick identified by the "candidatepick" attribute (above). If empty, the "candidate" child element defines the tag expression to use. Default: Empty.
- **prereqtarget:** (Optional) Set. Designates the container against which all pre-requisite tests need to be performed when determining the list of items available for selection. When displacement is utilized, pre-requisites need to be tested against the container to which the new picks will ultimately be added. Must be one of these values:
    - **container:** The default parent container is used.
    - **parent:** The next parent up the hierarchy is used, which parallels the corresponding displacement target.
    - **hero:** The top-level hero is used, which parallels the corresponding displacement target.
    - **Default:** "container".
- **empty:** (Optional) Text. Specifies the text message to be displayed if the user attempts to select an option and there are no available items to choose from. If empty, a default message is displayed. Default: Empty.
- **showfrozenfixed:** (Optional) Boolean. Indicates whether the table must be converted to a "fixed" table whenever the table is designated as frozen. Default: "no".
- **showfixedlast:** (Optional) Boolean. Indicates whether all non-deletable picks within the table are sorted to the end of the list of picks shown. Default: "no".
- **allowuserorder:** (Optional) Boolean. Indicates whether the items in the table can be re-ordered by the user. If enabled, the specified component must designate a suitable ordering field or a separate component with such a field must be specified via the "ordercomponent" attribute. Default: "no". **NOTE!** Verify that whatever sort set you use for showing the items includes the designated ordering field as its first sort key.
- **ordercomponent:** (Optional) Id. Specifies the unique id of an alternate component that possesses a suitable ordering field. This attribute is only applicable when the table supports user ordering. If empty, the ordering field is dictated by the component associated with the table. Default: Empty.
- **linkage:** (Optional) Id. Specifies the unique id of a thing that will be used as a linkage. When a new pick is added to the table, that pick has an automatic linkage setup to any existing pick derived from the specified thing. If no derived pick exists when the new pick is added, no linkage is ever created. If empty, no linkage is established. Default: Empty.
- **alwaysupdate:** (Optional) Boolean. Indicates whether the table must be dynamically updated after any modification to the actor so that the influence of other changes are always visually reflected to the user. Default: "no". **NOTE!** This option can significantly slow down display updates on slower computers, so only enable this if truly necessary.

The "table_dynamic" element also possesses child elements that define additional behaviors of the portal. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **list:** An optional "list" element may appear as defined by the given link. This element defines a List Tag Expression for the portal.
- **candidate:** An optional "candidate" element may appear as defined by the given link. This element defines a Candidate Tag Expression for the portal.
- **restriction:** An optional "restriction" element may appear as defined by the given link. This element defines a Restriction Tag Expression for the portal that identifies things which cannot be selected if they've already been added to this table.
- **needtag:** Zero or more "needtag" elements may appear as defined by the given link. This element defines a tag relationship that must exist between a prospective object and the container in order to list the object among the available items.
- **denytag:** Zero or more "denytag" elements may appear as defined by the given link. This element defines a tag relationship that must not exist between a prospective object and the container in order to list the object among the available items.
- **xacttag:** Zero or more "xacttag" elements may appear as defined by the given link. This element defines a tag that is assigned to the transaction pick while the choose form is visible.
- **secondary:** An optional "secondary" element may appear as defined by the given link. This element defines a Secondary Tag Expression that is associated with every new pick added via the portal.
- **existence:** An optional "existence" element may appear as defined by the given link. This element defines an Existence Tag Expression that is associated with every new pick added via the portal.
- **autotag:** Zero or more "autotag" elements may appear as defined by the given link. This element specifies tags that are autotag automatically assigned to each added thing.
- **chosen:** An optional "chosen" element may appear as defined by the given link. This element defines a Chosen Script for the portal.
- **titlebar:** An optional "titlebar" element may appear as defined by the given link. This element defines a TitleBar Script for the portal.

- **description:** An optional "description" element may appear as defined by the given link. This element defines a Description Script for the portal.
- **headertitle:** An optional "headertitle" element may appear as defined by the given link. This element defines a HeaderTitle Script for the portal.
- **additem:** An optional "additem" element may appear as defined by the given link. This element defines a AddItem Script for the portal.

THE "LIST" ELEMENT

The "list" element defines a List Tag Expression for the portal that limits the set of picks that are shown. Regardless of this tag expression, all picks added via this portal are always shown within it, enabling deletion of any object added through the table. The complete list of attributes for this element is below.

- **PCDATA:** TagExpr. Specifies the code comprising the List tag expression.

THE "CANDIDATE" ELEMENT

The "candidate" element defines a Candidate Tag Expression for the portal that limits the set of things/picks that are available for selection. If this element is omitted entirely, then the items available for selection must satisfy the List tag expression instead (above). The complete list of attributes for this element is below.

- **inheritlist:** (Optional) Boolean. Indicates whether the List tag expression (above) is automatically inherited into the Candidate inheritlist tag expression. If inherited, all available objects must satisfy both the Candidate tag expressions and the List tag expression. This eliminates the need to redundantly maintain the same filter logic within both tag expressions. If not inherited, then the Candidate tag expression supersedes the List tag expression. Default: "no".
- **PCDATA:** TagExpr. Specifies the code comprising the Candidate tag expression.

THE "RESTRICTION" ELEMENT

The "restriction" element defines a Restriction Tag Expression for the portal that further limits the set of things/picks that are available for selection. This tag expression is compared against all picks that have already been added to this table . Any object that already exists within the table is precluded from being selected again, resulting in it being omitted from the list of available choices. The complete list of attributes for this element is below.

- **PCDATA:** TagExpr. Specifies the code comprising the Restriction tag expression.

THE "NEEDTAG" ELEMENT

The "needtag" element defines a tag relationship that must exist between the object to be added and the prospective container. Tags from one tag group are enumerated within the container, then the object is tested to make sure that it has at least one matching tag with the same id in a separate tag group. If the tag is not found, the object is not valid for selection and omitted from the available list. The complete list of attributes for this element is below.

- **container:** Id. Specifies the unique id of the tag group to utilize within the container.
- **thing:** Id. Specifies the unique id of the tag group to check within the thing/pick.
- **usehero:** (Optional) Boolean. Indicates whether the container tags are pulled from the prospective container for the new pick or the hero. This distinction can be important when using displacement. Default: "no".

THE "DENYTAG" ELEMENT

The "denytag" element defines a tag relationship that must not exist between the object to be added and the prospective container. Tags from one tag group are enumerated within the container, then the object is tested to make sure that it does not possess any matching tags with the same ids in a separate tag group. If any matching tags are found, the object is not valid for selection and omitted from the available list. The complete list of attributes for this element is below.

- **container:** Id. Specifies the unique id of the tag group to utilize within the container.
- **thing:** Id. Specifies the unique id of the tag group to check within the thing/pick.
- **usehero:** (Optional) Boolean. Indicates whether the container tags are pulled from the prospective container for the new pick or the hero. This distinction can be important when using displacement. Default: "no".

THE "XACTTAG" ELEMENT

The "xacttag" element specifies a tag that is automatically added to the transaction pick while the choose form is shown. These tags allow you to indicate contextual information about where the buy template is being used so that you can tailor the behavior appropriately. The complete list of attributes for this element is below.

- **tag:** Id. Specifies the unique id of the tag to define within the tag group "transact".

THE "SECONDARY" ELEMENT

The "secondary" element defines a Secondary Tag Expression that is automatically associated with every new pick added via the portal. This new tag expression is treated like an additional Container Tag Expression for the pick that must also be satisfied. The complete list of attributes for this element is below.

- **phase:** (Optional) Id. Specifies the unique id of the evaluation phase during which the tag expression is tested. If empty, phase the default timing is used from the definition file. Default: Empty.
- **priority:** Integer. Specifies the evaluation priority during which the tag expression is tested. If empty, the default timing is used from the definition file. Default: Empty.
- **PCDATA:** TagExpr. Specifies the code comprising the Secondary tag expression.

THE "EXISTENCE" ELEMENT

The "existence" element defines an Existence Tag Expression that is automatically associated with every new pick added via the portal. If a pick ever fails to satisfy the tag expression during an evaluation cycle, the pick is automatically deleted. The complete list of attributes for this element is below.

- **phase:** (Optional) Id. Specifies the unique id of the evaluation phase during which the tag expression is tested. If empty, phase the default timing is used from the definition file. Default: Empty.
- **priority:** Integer. Specifies the evaluation priority during which the tag expression is tested. If empty, the default timing is used from the definition file. Default: Empty.
- **PCDATA:** TagExpr. Specifies the code comprising the Secondary tag expression.

THE "CHOSEN" ELEMENT

The "chosen" element defines a Chosen Script for the portal, which synthesizes the text to be displayed as the chosen item within the portal. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the Chosen script.

THE "TITLEBAR" ELEMENT

The "titlebar" element defines a TitleBar Script for the portal, which synthesizes the text to be displayed at the top of the choose form. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the TitleBar script.

THE "DESCRIPTION" ELEMENT

The "description" element defines a Description Script for the portal, which synthesizes the text to be displayed within the description region of the choose form for the currently selected item on the left. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the Description script.

THE "HEADERTITLE" ELEMENT

The "headertitle" element defines a HeaderTitle Script for the portal that synthesizes the text to be displayed at the top of the table as a header. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the HeaderTitle script.

THE "ADDITEM" ELEMENT

The "additem" element defines a AddItem Script for the portal that synthesizes the text to be displayed within the "add" item at the bottom of the table, where the user will click to add a new item. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the AddItem script.

EXAMPLE

The following example demonstrates what a dynamic table portal might look like. All default values are assumed for optional attributes.

```
<portal id="arMelee" style="tblNormal">
  <table_dynamic component="Gear"
    showtemplate="arWpnPick" choosetemplate="arWpnThing"
    buytemplate="BuyCash" selltemplate="SellCash">
   <list>component.WeapMelee</list>
   <candidate>!Equipment.Natural</candidate>
   <description/>
   <headertitle>
    @text = "Melee Weapons"
    </headertitle>
   <additem>
    @text = "Add New Melee Weapons"
    </additem>
   </table_dynamic>
  </portal>
```

TABLEAUTO ELEMENT (DATA)

THE "TABLE_AUTO" ELEMENT

Auto tables are a special kind of dynamic table that can only ever contain a single item. As such, an auto table looks like a dynamic table until the user clicks on the "add" item at the bottom. Instead of displaying a choose form, a new instance of a specific item is added to the table. This is extremely useful for tables of journal entries and character portraits. Each auto table is defined via the use of the "table_auto" element. The complete list of attributes for this element is below.

- **component:** Id. Specifies the unique id of the component that all shown objects must be derived from.
- **autothing:** Id. Specifies the unique id of the thing to be added to the table whenever the user clicks on the "add" item at the bottom.
- **showtemplate:** Id. Specifies the unique id of the template to be used for displaying the picks that have been added to the table.
- **showsortset:** (Optional) Id. Specifies the unique id of the sort set to be used for sequencing the items that exist within the table. If empty, all objects are sorted by name. Default: Empty.
- **showgaphorz:** (Optional) Integer. Specifies the gap along the horizontal axis to insert between items that exist within the table. Default: "0".
- **showgapvert:** (Optional) Integer. Specifies the gap along the vertical axis to insert between items that exist within the table. Default: "0".
- **columns:** (Optional) Integer. Specifies the number of columns of data to display within the table. Default: "1".
- **scrollable:** (Optional) Boolean. Indicates whether the table contents can be scrolled by the user. By default, a scroller is shown whenever the number of items exceeds the visible space, but you can disable this behavior. Default: "yes".
- **addtemplate:** (Optional) Id. Specifies the unique id of the template to be used for the "add" item that always appears at the bottom of the table and that users will click on to add an item to the table. This allows detailed controlled when the simple "additem" script is not sufficient. If empty, the "additem" element must be specified. Default: Empty.
- **addpick:** (Optional) Id. Specifies the unique id of a thing that is associated with the "add" item at the bottom of the table. HL will retrieve any pick based on this thing that exists within the target container and use it. This allows you to control what fields can be accessed from the "addtemplate". If empty, the "actor" pick is used, except within a gizmo, where its "defaultthing" is used instead. Default: Empty.
- **addspace:** (Optional) Integer. Specifies the additional vertical space to be inserted when displaying the simple "add" item at the bottom of the table by using the "additem" script. The height of the item is based on the font height of the text shown, so this attribute allows you to insert additional padding if you wish. Default: "2".
- **headertemplate:** (Optional) Id. Specifies the unique id of the template to be used for a header item that appears at the top of the table. This allows you to add column headers above various pieces of information in the table. If empty, the "headertitle" element dictates whether a header is displayed above the table. Default: Empty.
- **headerpick:** (Optional) Id. Specifies the unique id of a thing that is associated with the header item at the top of the table. HL will retrieve any pick based on this thing that exists within the target container and use it. This allows you to control what fields can be accessed from the "headertemplate". If empty, the "actor" pick is used, except within a gizmo, where its "defaultthing" is used instead. Default: Empty.
- **showfrozenfixed:** (Optional) Boolean. Indicates whether the table must be converted to a "fixed" table whenever the table is designated as frozen. Default: "no".
- **allowuserorder:** (Optional) Boolean. Indicates whether the items in the table can be re-ordered by the user. If enabled, the specified component must designate a suitable ordering field or a separate component with such a field must be specified via the "ordercomponent" attribute. Default: "no". **NOTE!** Verify that whatever sort set you use for showing the items includes the designated ordering field as its first sort key.
- **ordercomponent:** (Optional) Id. Specifies the unique id of an alternate component that possesses a suitable ordering field. This attribute is only applicable when the table supports user ordering. If empty, the ordering field is dictated by the component associated with the table. Default: Empty.
- **linkage:** (Optional) Id. Specifies the unique id of a thing that will be used as a linkage. When a new pick is added to the table, that pick has an automatic linkage setup to any existing pick derived from the specified thing. If no derived pick exists when the new pick is added, no linkage is ever created. If empty, no linkage is established. Default: Empty.
- **alwaysupdate:** (Optional) Boolean. Indicates whether the table must be dynamically updated after any modification to the actor so that the influence of other changes are always visually reflected to the user. Default: "no". **NOTE!** This option can significantly slow down display updates on slower computers, so only enable this if truly necessary.

The "table_auto" element also possesses child elements that define additional behaviors of the portal. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **list:** An optional "list" element may appear as defined by the given link. This element defines a List Tag Expression for the portal.
- **secondary:** An optional "secondary" element may appear as defined by the given link. This element defines a Secondary Tag Expression that is associated with every new pick added via the portal.
- **existence:** An optional "existence" element may appear as defined by the given link. This element defines an Existence Tag Expression that is associated with every new pick added via the portal.

- **autotag:** Zero or more "autotag" elements may appear as defined by the given link. This element specifies tags that are automatically assigned to each added thing.
- **headertitle:** An optional "headertitle" element may appear as defined by the given link. This element defines a HeaderTitle Script for the portal.
- **additem:** An optional "additem" element may appear as defined by the given link. This element defines a AddItem Script for the portal.

THE "LIST" ELEMENT

The "list" element defines a List Tag Expression for the portal that limits the set of picks that are shown. Regardless of this tag expression, all picks added via this portal are always shown within it, enabling deletion of any object added through the table. The complete list of attributes for this element is below.

- **PCDATA:** TagExpr. Specifies the code comprising the List tag expression.

THE "SECONDARY" ELEMENT

The "secondary" element defines a Secondary Tag Expression that is automatically associated with every new pick added via the portal. This new tag expression is treated like an additional Container Tag Expression for the pick that must also be satisfied. The complete list of attributes for this element is below.

- **phase:** (Optional) Id. Specifies the unique id of the evaluation phase during which the tag expression is tested. If empty, the default timing is used from the definition file. Default: Empty.
- **priority:** Integer. Specifies the evaluation priority during which the tag expression is tested. If empty, the default timing is used from the definition file. Default: Empty.
- **PCDATA:** TagExpr. Specifies the code comprising the Secondary tag expression.

THE "EXISTENCE" ELEMENT

The "existence" element defines an Existence Tag Expression that is automatically associated with every new pick added via the portal. If a pick ever fails to satisfy the tag expression during an evaluation cycle, the pick is automatically deleted. The complete list of attributes for this element is below.

- **phase:** (Optional) Id. Specifies the unique id of the evaluation phase during which the tag expression is tested. If empty, the default timing is used from the definition file. Default: Empty.
- **priority:** Integer. Specifies the evaluation priority during which the tag expression is tested. If empty, the default timing is priority used from the definition file. Default: Empty.
- **PCDATA:** TagExpr. Specifies the code comprising the Secondary tag expression.

THE "HEADERTITLE" ELEMENT

The "headertitle" element defines a HeaderTitle Script for the portal that synthesizes the text to be displayed at the top of the table as a header. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the HeaderTitle script.

THE "ADDITEM" ELEMENT

The "additem" element defines a AddItem Script for the portal that synthesizes the text to be displayed within the "add" item at the bottom of the table, where the user will click to add a new item. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the AddItem script.

EXAMPLE

The following example demonstrates what an auto table portal might look like. All default values are assumed for optional attributes.

```
<portal id="peImages" style="tblNormal">
  <table_auto component="UserImage"
     showtemplate="peImage" autothing="mscUserImg">
   <headertitle>
     @text = "Gallery"
     </headertitle>
   <additem>
     @text = "Add Another Image"
     </additem>
   </table_auto>
  </portal>
```

SETTINGEDIT ELEMENT (DATA)

THE "SETTING_EDIT" ELEMENT

The "setting_edit" element is used in a single location within HL. This portal corresponds to the button that allows the user to edit the various Settings on the Configure Hero form. It's purpose is to allow you to position the button where you want it on the Configure Hero form. There are no attributes or child elements for this element.

EXAMPLE

The following example demonstrates what a settingedit portal might look like. All default values are assumed for optional attributes.

```
<portal id="cnfEdit" style="special"
      tiptext="Click here to change the settings governing your character.">
   <setting_edit/>
   </portal>
```

SETTINGSUMMARY ELEMENT (DATA)

THE "SETTING_SUMMARY" ELEMENT

The "setting_summary" element is used in a single location within HL. This portal corresponds to the table that displays the currently selected Settings on the Configure Hero form. It's purpose is to allow you to position and size the table on the Configure Hero form. There are no attributes or child elements for this element.

EXAMPLE

The following example demonstrates what a setting_summary portal might look like. All default values are assumed for optional attributes.

```
<portal id="cnfSummary" style="special">
   <setting_summary/>
   </portal>
```

ALLIANCE ELEMENT (DATA)

THE "ALLIANCE" ELEMENT

The "alliance" element is used in a single location within HL. This portal corresponds to the menu that allows the user to toggle whether an actor is an ally or enemy on the Configure Hero form. It's purpose is to allow you to position and size the menu on the Configure Hero form. There are no attributes or child elements for this element.

EXAMPLE

The following example demonstrates what a alliance portal might look like. All default values are assumed for optional attributes.

```
<portal id="cnfAlly" style="special">
   <alliance/>
   </portal>
```

OUTPUTLABEL ELEMENT (DATA)

THE "OUTPUT_LABEL" ELEMENT

The role of the "output_label" element is identical to the "label" element, except that it is designed for use exclusively within character sheet output. Any data that you want to display within a sheet (i.e. most everything within a sheet) will require the use of an "output_label" element. The complete list of attributes for this element is below.

IMPORTANT! Only one mechanism for specifying the label contents may be employed within a given output label portal. That means you may use either the "text" attribute, the "field" attribute, or the "labeltext" script to define the contents. Use of multiple mechanisms will result in a compilation error.

- **text:** (Optional) Text. Specifies a string of literal text to be displayed within the label. Default: Empty.
- **field:** (Optional) Id. Specifies the unique id of the field whose value is to be displayed within the label. The field must exist within the pick/thing associated with the containing template. If this portal is not defined within a template, a field-based label is not allowed. Default: Empty.
- **ismultiline:** (Optional) Boolean. Indicates whether the label text is to be treated as multi-line or merely a single line of output. Default: "no".

The "output_label" element also possesses child elements that define additional behaviors of the portal. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **labeltext:** An optional "labeltext" element may appear as defined by the given link. This element defines a Label Script that is used for synthesizing the text to be output.

THE "LABELTEXT" ELEMENT

The "labeltext" element defines a Label Script for the portal. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the Label script.

EXAMPLE

The following example demonstrates what an output label portal might look like. All default values are assumed for optional attributes.

```
<portal id="name" style="outNameLg">
  <output_label field="name"/>
  </portal>

<portal id="oHeroName" style="outHeroNam">
  <output_label>
   <labeltext>
    @text = hero.actorname
    var result as number
    result = compare(@text, "")
    if (result = 0) then
     @text = "- Unnamed Hero -"
     endif
    </labeltext>
   </output_label>
  </portal>
```

OUTPUTIMAGE ELEMENT (DATA)

THE "IMAGE_FIELD" ELEMENT

The role of the "output_image" element is an amalgam of the various "image" elements, except that it is designed for use exclusively within character sheet output. Any images that you want to display within a sheet will require the use of an "output_image" element. The complete list of attributes for this element is below.

IMPORTANT! Only one mechanism for specifying the image contents may be employed within a given output image portal. That means you may use either the "image" attribute or the "field" attribute to define the contents. Use of multiple mechanisms will result in a compilation error.

- **image:** (Optional) Text. Specifies the filename of a bitmap image within the data file folder for the game system. The given image is displayed within the portal.
- **field:** (Optional) Id. Specifies the unique id of the field whose contents dictate the image to to be displayed within the portal. The field may contain a user-defined image, a reference image, or a filename of a bitmap image within the data file folder for the game system. The field must exist within the pick/thing associated with the containing template. If this portal is not defined within a template, a field-based image is not allowed.
- **istransparent:** (Optional) Boolean. Indicates whether the image should be treated as transparent, wherein the pixel color at position 0,0 is considered transparent throughout the image. Default: "no".

EXAMPLE

The following example demonstrates what an output image portal might look like. All default values are assumed for optional attributes.

```
<portal id="oHLLogo" style="outNormal">
  <output_image image="sheet_hllogo.bmp"/>
  </portal>

<portal id="image" style="outNormal">
  <output_image field="acTacImage"/>
  </portal>
```

OUTPUTTABLE ELEMENT (DATA)

THE "OUTPUT_TABLE" ELEMENT

Output tables are exclusively for use within character sheet output, hence the name. Due to their behavior, output tables are very similar to fixed tables, with the primary distinction being that output tables can support variable height items. Each output table is defined via the use of the "output_table" element. The complete list of attributes for this element is below.

- **component:** Id. Specifies the unique id of the component that all shown objects must be derived from.
- **showtemplate:** Id. Specifies the unique id of the template to be used for displaying the picks that have been added to the table.
- **showpicks:** (Optional) Set. Designates the source from which the picks shown are retrieved from. Must be one of these values:

    - **container:** The picks shown are from the implicitly identified container. If the containing scene is a form associated with a gizmo, the gizmo is used, else the actor is used.
    - **hero:** The picks shown are from the active actor.
    - **actor:** The picks shown represent all actors in the entire portfolio.
    - **lead:** The picks shown represent all lead actors in the entire portfolio.
    - **minion:** The picks shown are all immediate minions for the active actor.
    - **Default:** "container".
- **showsortset:** (Optional) Id. Specifies the unique id of the sort set to be used for sequencing the items that exist within the table. If empty, all objects are sorted by name. Default: Empty.
- **showgaphorz:** (Optional) Integer. Specifies the gap along the horizontal axis to insert between items that exist within the table. Default: "0".
- **showgapvert:** (Optional) Integer. Specifies the gap along the vertical axis to insert between items that exist within the table. Default: "0".
- **columns:** (Optional) Integer. Specifies the number of columns of data to display within the table. Default: "1".
- **varyheight:** (Optional) Boolean. Indicates whether the items output within the table can be of varying height. This is extremely valuable for material like special abilities, journal entries, etc. Such material can have description text that ranges from short to long, and it's necessary that each item in the table have a suitable height that matches its contents. Default: "no".
- **headertemplate:** (Optional) Id. Specifies the unique id of the template to be used for a header item that appears at the top of the table. This allows you to add column headers above various pieces of information in the table. If empty, the "headertitle" element dictates whether a header is displayed above the table. Default: Empty.
- **headerpick:** (Optional) Id. Specifies the unique id of a thing that is associated with the header item at the top of the table. HL will retrieve any pick based on this thing that exists within the target container and use it. This allows you to control what fields can be accessed from the "headertemplate". If empty, the "actor" pick is used, except within a gizmo, where its "defaultthing" is used instead. Default: Empty.

The "output_table" element also possesses child elements that define additional behaviors of the portal. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **list:** An optional "list" element may appear as defined by the given link. This element defines a List Tag Expression for the portal.
- **headertitle:** An optional "headertitle" element may appear as defined by the given link. This element defines a HeaderTitle Script for the portal.

THE "LIST" ELEMENT

The "list" element defines a List Tag Expression for the portal that limits the set of picks that are shown. The complete list of attributes for this element is below.

- **PCDATA:** TagExpr. Specifies the code comprising the List tag expression.

THE "HEADERTITLE" ELEMENT

The "headertitle" element defines a HeaderTitle Script for the portal that synthesizes the text to be displayed at the top of the table as a header. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the HeaderTitle script.

EXAMPLE

The following example demonstrates what an output table portal might look like. All default values are assumed for optional attributes.

```
<portal id="oJournTbl" style="outNormal">
  <output_table component="Journal" varyheight="yes"
    showtemplate="oJrnPick" showpicks="yes">
  </output_table>
  </portal>

<portal id="oAdjust" style="outNormal">
  <output_table component="Adjustment" showtemplate="oAdjPick" columns="2">
   <list>Helper.Activated</list>
   <headertitle>
    @text = "Activated Adjustments"
    </headertitle>
  </output_table>
  </portal>
```

OUTPUTDOTS ELEMENT (DATA)

THE "OUTPUT_DOTS" ELEMENT

The role of the "output_dots" element is a special purpose "output_label" portal that it is designed for use exclusively within character sheet output. This portal makes it possible to easily insert a series of alternating dots and spaces between two portals within character sheet output. The dots are guaranteed to be vertically aligned, regardless of the span over which they cover, ensuring that a sequence of items in a table that use the dot spanning will look clean and consistent. This portal behaves just like an output_label portal in other respects, so be sure to assign a style the provides the font in which to render the dot sequence. There are no attributes or child elements for this portal.

EXAMPLE

The following example demonstrates what an output dots portal might look like. All default values are assumed for optional attributes.

```
<portal id="dots" style="outPlain">
  <output_dots/>
  </portal>
```

OUTPUTSEPARATOR ELEMENT (DATA)

THE "OUTTPUT_SEPARATOR" ELEMENT

The role of the "output_separator" element is a special separator portal for use within sheet output. The portal inserts a vertical or horizontal bar of solid black between groupings of portals, acting as a visual separator between them. The complete list of attributes for this element is below.

- **isvertical:** (Optional) Boolean. Indicates whether the separator should be drawn horizontally or vertically. Default: "no".

EXAMPLE

The following example demonstrates what an output separator portal might look like. All default values are assumed for optional attributes.

```
<portal id="separator" style="oSeparator">
  <output_separator isvertical="no"/>
  </portal>
```

TEMPLATE ELEMENT (DATA)

THE "TEMPLATE" ELEMENT

Templates are collection of related portals that are all linked to a common thing or pick. Each template is specified through the use of a "template" element. The complete list of attributes for this element is below.

- **id:** Id. Specifies the unique id of the template. This id is used in all references to the template.
- **name:** Text. Public name associated with the template. Maximum length of 50 characters.
- **compset:** Id. Specifies the unique id of the component set that things/picks shown within the template derive from.
- **width:** (Optional) Integer. Specifies the initial width to be utilized for the template. If empty, the width is automatically determined by a set of rules. Default: Empty.
- **height:** (Optional) Integer. Specifies the initial height to be utilized for the template. Default: "100".
- **marginhorz:** (Optional) Integer. Specifies the margin gap included at both ends along the horizontal axis. The usable width of the template is the actual width minus double the horizontal margin. Default: "0".
- **marginvert:** (Optional) Integer. Specifies the margin gap included at both ends along the vertical axis. The usable height of the template is the actual height minus double the vertical margin. Default: "0".
- **istransaction:** (Optional) Boolean. Indicates whether this template is intended for use with buy or sell transactions. Default: "no".

The "template" element also possesses child elements that define various facets of the template. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **live:** An optional "live" element may appear as defined by the given link. This element defines a Live Tag Expression that determines whether the template is shown to the user.
- **portal:** One or more "portal" elements must appear as defined by the given link. This element specifies the individual portals which exist within the template.
- **position:** An optional "position" element may appear as defined by the given link. This element defines a Position Script through which the portals are sized and positioned within the template.
- **header:** An optional "header" element may appear as defined by the given link. This element defines a Header Script that coordinates the sizing and positioning of portals used within the header for the template.

THE "LIVE" ELEMENT

The "live" element defines a Live Tag Expression for the template, which determines whether the template is shown. The tag expression is compared against the tags assigned to the container associated with the template (e.g. the actor). If the tag expression is satisfied, the template is visible, else it is hidden. The complete list of attributes for this element is below.

- **PCDATA:** TagExpr. Specifies the code comprising the Live tag expression.

THE "POSITION" ELEMENT

The "position" element defines a Position Script for the template, which performs all the appropriate sizing and positioning of the contained portals within the region of the template. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the Position script.

THE "HEADER" ELEMENT

The "header" element defines a Header Script for the template, which works exactly like the Position script, except for the portals it manipulates. This script is only used if the template is used as a dual header and contains specifically designated header portals. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the Header script.

EXAMPLE

The following example demonstrates what a "template" element might look like. All default values are assumed for optional attributes.

```
<template id="baAttrPick" name="Attribute Pick" compset="Attribute"
    marginhorz="18" marginvert="9">
  <portal id="name" style="lblXLarge" showinvalid="yes">
   <label field="name"/>
   </portal>
  <portal id="value" style="incrSimple">
   <incrementer field="trtUser"/>
   <mouseinfo mousepos="middle+above">
     @text = "Adjust attribute by clicking on the arrows."
     </mouseinfo>
   </portal>
  <portal id="info" style="actInfo">
   <action action="info"/>
   </portal>
  <position><![CDATA[
   ~set up our height based on our tallest portal
   height = portal[info].height

   ~if this is a "sizing" calculation, we're done
   if (issizing <> 0) then
     done
     endif

   ~position our tallest portal at the top
   portal[info].top = 0

   ~center the other portals vertically
   perform portal[name].centervert
   perform portal[value].centervert

   ~position the info portal on the far right
   perform portal[info].alignedge[right,0]

   ~position the incrementer to the left of the info portal (plus a gap)
   perform portal[value].alignrel[rtol,info,-10]

   ~position the name on the left and limit its width to available space
   portal[name].left = 0
   portal[name].width = minimum(portal[name].width,portal[value].left - portal[name].left - 10)
   </position>
  </template>
```

LAYOUT ELEMENT (DATA)

THE "LAYOUT" ELEMENT

Layouts contain an assortment of portals and templates that serve a unified purpose when presenting information to the user. Each layout is specified through the use of a "layout" element. The complete list of attributes for this element is below.

- **id:** Id. Specifies the unique id of the layout. This id is used in all references to the layout.
- **marginhorz:** (Optional) Integer. Specifies the margin gap included at both ends along the horizontal axis. The usable width of the layout is the actual width minus double the horizontal margin. Default: "0".
- **marginvert:** (Optional) Integer. Specifies the margin gap included at both ends along the vertical axis. The usable height of the layout is the actual height minus double the vertical margin. Default: "0".

The "layout" element also possesses child elements that define various facets of the layout. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **live:** An optional "live" element may appear as defined by the given link. This element defines a Live Tag Expression that determines whether the template is shown to the user.
- **portalref:** Zero or more "portalref" elements may appear as defined by the given link. This element specifies the individual portals which are utilized within the layout.
- **templateref:** Zero or more "templateref" elements may appear as defined by the given link. This element specifies the individual templates which are utilized within the layout.
- **position:** An optional "position" element may appear as defined by the given link. This element defines a Position Script through which the portals are sized and positioned within the template.

THE "LIVE" ELEMENT

The "live" element defines a Live Tag Expression for the layout, which determines whether the layout is shown. The tag expression is compared against the tags assigned to the container associated with the layout (e.g. the actor). If the tag expression is satisfied, the layout is visible, else it is hidden. The complete list of attributes for this element is below.

- **PCDATA:** TagExpr. Specifies the code comprising the Live tag expression.

THE "PORTALREF" ELEMENT

The "portalref" element identifies a portal used within the layout and assigns it a logical name for use within the Position script. The complete list of attributes for this element is below.

- **portal:** Id. Unique id of the portal to be utilized within the layout.
- **reference:** (Optional) Id. Specifies the alternate unique id to be used when referencing this portal within the Position script. This makes it possible to re-use the same portal more than once within the same layout, giving each instance a different logical id. If empty, the reference id is simply the portal id. Default: Empty.
- **taborder:** (Optional) Integer. Specifies the relative tab order position of the portal and its contents within the overall layout. As long as the tab orders are different for all portals and templates within the layout, HL can properly sequence the flow of control. Default: "0".

THE "TEMPLATEREF" ELEMENT

The "templateref" element identifies a template used within the layout and assigns it a logical name for use within the Position script. The complete list of attributes for this element is below.

- **template:** Id. Unique id of the template to be utilized within the layout.
- **reference:** (Optional) Id. Specifies the alternate unique id to be used when referencing this template within the Position script. This makes it possible to re-use the same template more than once within the same layout, giving each instance a different logical id. If empty, the reference id is simply the template id. Default: Empty.
- **thing:** (Optional) Id. Specifies the unique id of the thing or pick that is associated with this template. The given object is used with the template when determining the contents of fields and such. If empty, no thing/pick is associated with the template, which means that the template cannot contain any portals that utilize a field reference. Default: Empty.
- **ispick:** (Optional) Boolean. Indicates whether the template is associated with a pick or a thing. If a pick, the first pick within the container that based on the specified thing is used. Default: "yes".
- **taborder:** (Optional) Integer. Specifies the relative tab order position of the template and its contents within the overall layout. As long as the tab orders are different for all portals and templates within the layout, HL can properly sequence the flow of control. Default: "0".

THE "POSITION" ELEMENT

The "position" element defines a Position Script for the layout, which performs all the appropriate sizing and positioning of the contained portals and templates within the region of the layout. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the Position script.

EXAMPLE

The following example demonstrates what a "layout" element might look like. All default values are assumed for optional attributes.

```
<layout id="journal">
  <portalref portal="jrTitle"/>
  <portalref portal="journal" taborder="10"/>
  <templateref template="jrHeader" thing="actor"/>
  <position>
   ~configure whether our header is visible or not
   ~Note: If the header is non-visible, no space is allotted for it below.
   template[jrHeader].visible = 1


   ~position the title at the top, followed by the template, and then the table
   perform portal[jrTitle].autoplace
   perform template[jrHeader].autoplace[6]
   perform portal[journal].autoplace[9]
   </position>
  </layout>
```

PANEL ELEMENT (DATA)

THE "PANEL" ELEMENT

Panels contain one or more layouts and orchestrate the presentation of material for a specific region within HL. Panels are used to define the contents of the various tabs shown across the top for editing character data, as well as for summary panels shown on the right. Each panel is specified through the use of a "panel" element. The complete list of attributes for this element is below.

- **id:** Id. Specifies the unique id of the panel. This id is used in all references to the panel.
- **name:** Text. Public name associated with the panel. Maximum length of 50 characters.
- **order:** (Optional) Integer. Specifies the sequence in which this panel is displayed to the user. For a tab-based panel, this value controls the order in which the tabs are shown across the top. For a summary panel, this value controls the default order in which the summary panels are shown. Default: "0".
- **issumary:** (Optional) Boolean. Indicates whether this panel is intended for use as a summary panel or a tab-based panel. Default: "no".
- **marginhorz:** (Optional) Integer. Specifies the margin gap included at both ends along the horizontal axis. The usable width of the panel is the actual width minus double the horizontal margin. Default: "0".
- **marginvert:** (Optional) Integer. Specifies the margin gap included at both ends along the vertical axis. The usable height of the panel is the actual height minus double the vertical margin. Default: "0".
- **underlay:** (Optional) Text. Specifies the filename of a bitmap image that is blitted on top of the background and beneath all other visual elements within the panel. This bitmap file is assumed to reside within the data file folder for the game system. If empty, no underlay bitmap is utilized. Default: Empty.

The "panel" element also possesses child elements that define various facets of the panel. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **live:** An optional "live" element may appear as defined by the given link. This element defines a Live Tag Expression that determines whether the panel is shown to the user.
- **layoutref:** Zero or more "layoutref" elements may appear as defined by the given link. This element specifies the individual layouts which are utilized within the panel.
- **position:** An optional "position" element may appear as defined by the given link. This element defines a Position Script through which the contained layouts are sized and positioned within the panel.

THE "LIVE" ELEMENT

The "live" element defines a Live Tag Expression for the panel, which determines whether the panel is shown. The tag expression is compared against the tags assigned to the container associated with the panel (e.g. the actor). If the tag expression is satisfied, the panel is visible, else it is hidden. For tab panels, hidden implies that the tab itself does not appear. The complete list of attributes for this element is below.

- **PCDATA:** TagExpr. Specifies the code comprising the Live tag expression.

THE "LAYOUTREF" ELEMENT

The "layoutref" element identifies a layout used within the panel and assigns it a logical name for use within the Position script. The complete list of attributes for this element is below.

- **layout:** Id. Unique id of the layout to be utilized within the panel.
- **reference:** (Optional) Id. Specifies the alternate unique id to be used when referencing this layout within the Position script. This makes it possible to re-use the same layout more than once within the same panel, giving each instance a different logical id. If empty, the reference id is simply the layout id. Default: Empty.

THE "POSITION" ELEMENT

The "position" element defines a Position Script for the panel, which performs all the appropriate sizing and positioning of the contained layouts within the panel. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the Position script.

EXAMPLE

The following example demonstrates what a "panel" element might look like. All default values are assumed for optional attributes.

```
<panel id="basics" name="Basics" order="110"
    marginhorz="5" marginvert="5">
  <live>!HideTab.basics</live>
  <layoutref layout="basics"/>
  <position>
    ~script code goes here
    </position>
  </panel>
```

FORM ELEMENT (DATA)

THE "FORM" ELEMENT

Forms contain one or more layouts and orchestrate the presentation of material within a standalone form within HL. Forms are used to define the contents of the Configure Hero form, the Tactical Console, and other visual pieces. Each form is specified through the use of a "form" element. The complete list of attributes for this element is below.

- **id:** Id. Specifies the unique id of the form. This id is used in all references to the form.
- **name:** Text. Public name associated with the form. Maximum length of 50 characters.
- **showbutton:** (Optional) Boolean. Indicates whether a button should be added at the bottom of the form that allows the user to close the form. Default: "yes".
- **entity:** (Optional) Id. Specifies the unique of an entity for which this form is designed to edit the contents. If empty, the form cannot be used for editing any gizmos. Default: Empty.
- **marginhorz:** (Optional) Integer. Specifies the margin gap included at both ends along the horizontal axis. The usable width of the panel is the actual width minus double the horizontal margin. Default: "0".
- **marginvert:** (Optional) Integer. Specifies the margin gap included at both ends along the vertical axis. The usable height of the panel is the actual height minus double the vertical margin. Default: "0".
- **defwidth:** (Optional) Integer. Specifies the default initial width to be used for the form. Default: "300".
- **defheight:** (Optional) Integer. Specifies the default initial height to be used for the form. Default: "300".
- **minwidth:** (Optional) Integer. Specifies the minimum width to be enforced for the form. Default: "0".
- **minheight:** (Optional) Integer. Specifies the minimum height to be enforced for the form. Default: "0".
- **maxwidth:** (Optional) Integer. Specifies the maximum width to be enforced for the form. Default: "0".
- **maxheight:** (Optional) Integer. Specifies the maximum height to be enforced for the form. Default: "0".

The "form" element also possesses child elements that define various facets of the form. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **layoutref:** Zero or more "layoutref" elements may appear as defined by the given link. This element specifies the individual layouts which are utilized within the form.
- **position:** An optional "position" element may appear as defined by the given link. This element defines a Position Script through which the contained layouts are sized and positioned within the form.

THE "LAYOUTREF" ELEMENT

The "layoutref" element identifies a layout used within the form and assigns it a logical name for use within the Position script. The complete list of attributes for this element is below.

- **layout:** Id. Unique id of the layout to be utilized within the form.
- **reference:** (Optional) Id. Specifies the alternate unique id to be used when referencing this layout within the Position script. This makes it possible to re-use the same layout more than once within the same form, giving each instance a different logical id. If empty, the reference id is simply the layout id. Default: Empty.

THE "POSITION" ELEMENT

The "position" element defines a Position Script for the form, which performs all the appropriate sizing and positioning of the contained layouts within the form. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the Position script.

EXAMPLE

The following example demonstrates what a "form" element might look like. All default values are assumed for optional attributes.

```
<panel id="config" name="Configure">
  <layoutref layout="configure"/>
  <position>
   ~render the layout to generate its dimensions
   perform layout[configure].render
   ~set the width and height of the form to the dimensions of the layout
   width = layout[configure].width
   height = layout[configure].height
   </position>
  </panel>
```

SHEET ELEMENT (DATA)

THE "SHEET" ELEMENT

Sheets contain one or more layouts and orchestrate the presentation of material within a specific page of a character sheet. Each sheet is specified through the use of a "sheet" element. The complete list of attributes for this element is below.

- **id:** Id. Specifies the unique id of the sheet. This id is used in all references to the sheet.
- **name:** Text. Public name associated with the sheet. Maximum length of 50 characters.
- **landscape:** (Optional) Boolean. Indicates whether the sheet is to be output in landscape or portrait orientation. Default: "no".
- **spillover:** Optional) Boolean. Indicates whether this sheet utilized spillover behaviors or not. If spillover behavior is enabled, the sheet is output repeatedly until all objects rendered via the page have been output. Default: "no".

The "sheet" element also possesses child elements that define various facets of the sheet. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **live:** An optional "live" element may appear as defined by the given link. This element defines a Live Tag Expression that determines whether the sheet is actually output for the character.
- **layoutref:** Zero or more "layoutref" elements may appear as defined by the given link. This element specifies the individual layouts which are utilized within the sheet.
- **position:** An optional "position" element may appear as defined by the given link. This element defines a Position Script through which the contained layouts are sized and positioned within the sheet.

THE "LIVE" ELEMENT

The "live" element defines a Live Tag Expression for the sheet, which determines whether the sheet is included in the output. The tag expression is compared against the tags assigned to the actor being output. If the tag expression is satisfied, the sheet is output, else it is omitted. The complete list of attributes for this element is below.

- **PCDATA:** TagExpr. Specifies the code comprising the Live tag expression.

THE "LAYOUTREF" ELEMENT

The "layoutref" element identifies a layout used within the sheet and assigns it a logical name for use within the Position script. The complete list of attributes for this element is below.

- **layout:** Id. Unique id of the layout to be utilized within the sheet.
- **reference:** (Optional) Id. Specifies the alternate unique id to be used when referencing this layout within the Position script. This makes it possible to re-use the same layout more than once within the same sheet, giving each instance a different logical id. If empty, the reference id is simply the layout id. Default: Empty.

THE "POSITION" ELEMENT

The "position" element defines a Position Script for the sheet, which performs all the appropriate sizing and positioning of the contained layouts within the sheet. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the Position script.

EXAMPLE

The following example demonstrates what a "sheet" element might look like. All default values are assumed for optional attributes.

```
<sheet id="standard2" name="Standard character sheet, page #2" spillover="yes">
  <layoutref layout="oStandard2" reference="left"/>
  <layoutref layout="oStandard2" reference="right"/>
  <position>
   ~setup the gap to be used between the various sections of the character sheet
   autogap = 40
   global[sectiongap] = autogap

   ~calculate the width of the two columns of the character sheet, leaving a
    ~suitable center gap between them
    var colwidth as number
    colwidth = (width - 50) / 2

   ~output the layout on the lefthand side with whatever information will fit
    layout[left].width = colwidth
    layout[left].height = height
    perform layout[left].render

   ~output the layout on the righthand side with whatever information will fit
    layout[right].width = colwidth
    layout[right].height = height
    perform layout[right].render
    </position>
  </sheet>
```

DOSSIER ELEMENT (DATA)

THE "DOSSIER" ELEMENT

Dossiers represent a logical set of output for a portfolio, such as a character sheet, statblock, or even export data for use with another product. Each dossier is specified through the use of a "dossier" element. The complete list of attributes for this element is below.

- **id:** Id. Specifies the unique id of the dossier. This id is used in all references to the dossier. When the dossier is being rendered, a global tag "dossier.<this id>" is added to the portfolio allow you to see which dossier is being rendered.
- **name:** Text. Public name associated with the dossier. Maximum length of 100 characters.

The "dossier" element also possesses child elements that define various facets of the dossier. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

IMPORTANT! Exactly one of these child elements may be specified for each dossier. If multiple are given, a compiler error will be reported. The chosen child element dictates the type of dossier that is being defined and its characteristics.

- **dossier_sheet:** An optional "dossier_sheet" element may appear as defined by the given link. This element defines the structure of a printed dossier, such as a character sheet.
- **dossier_text:** An optional "dossier_text" element may appear as defined by the given link. This element specifies text-based output, such as a statblock.
- **dossier_export:** An optional "dossier_export" element may appear as defined by the given link. This element defines output that is tailored for use within another product, such as d20Pro.

THE "DOSSIER_SHEET" ELEMENT

The "dossier_sheet" element defines the structure of a printed dossier, containing a collection of sheets. The complete list of attributes for this element is below.

- **grouping:** Text. Specifies the name of the group into which this dossier should be placed for display. The groupings are used to logically organize multiple dossiers for presentation to the user, with the various groupings being sorted alphabetically.
- **default:** (Optional) Boolean. Indicates whether this dossier should be the default dossier that is pre-selected for the user when first outputting a printed dossier. Default: "no".

The "dossier_sheet" element also possesses child elements that describe its contents. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **live:** An optional "live" element may appear as defined by the given link. This element defines a Live Tag Expression that determines whether the dossier is made available for selection.
- **sheetref:** Zero or more "sheetref" elements may appear as defined by the given link. This element identifies a sheet to be included in the dossier output.

THE "LIVE" ELEMENT

The "live" element defines a Live Tag Expression for the dossier, which determines whether the dossier is shown to the user as a selection. The tag expression is compared against the tags assigned to the current actor. If the tag expression is satisfied, the dossier can be output, else it is hidden. This is ideal for situations like spellbooks that should only be shown for spellcasters. The complete list of attributes for this element is below.

- **PCDATA:** TagExpr. Specifies the code comprising the Live tag expression.

THE "SHEETREF" ELEMENT

The "sheetref" element specifies a sheet to be output as part of the dossier. The sequence in which the sheets are specified dictates the order in which they will be processed in the output. The complete list of attributes for this element is below.

- **sheet:** Id. Specifies the unique id of the sheet to be included in the dossier output.

THE "DOSSIER_TEXT" ELEMENT

The "dossier_text" element defines the logic used in outputting a text-based dossier, such as a statblock. The complete list of attributes for this element is below.

- **grouping:** Text. Specifies the name of the group into which this dossier should be placed for display. The groupings are used to logically organize multiple dossiers for presentation to the user, with the various groupings being sorted alphabetically.
- **styles:** (Optional) Text. Specifies the output styles that are supported for this dossier. The various output styles you can specify are "plain", "html", and "bbcode". You can combine multiple styles by placing a '+' between them (e.g. "plain+bbcode"). If empty, the "plain" style is assumed. Default: Empty.
- **default:** (Optional) Boolean. Indicates whether this dossier should be the default dossier that is pre-selected for the user when first outputting a text dossier. Default: "no".

The "dossier_text" element also possesses child elements that describe its contents. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **live:** An optional "live" element may appear as defined by the given link. This element defines a Live Tag Expression that determines whether the dossier is made available for selection.
- **synthesize:** Zero or more "synthesize" elements may appear as defined by the given link. This element defines the Synthesize Script used to generate the text to be output.

THE "LIVE" ELEMENT

The "live" element defines a Live Tag Expression for the dossier, which determines whether the dossier is shown to the user as a selection. The tag expression is compared against the tags assigned to the current actor. If the tag expression is satisfied, the dossier can be output, else it is hidden. This is ideal for situations like spellbooks that should only be shown for spellcasters. The complete list of attributes for this element is below.

- **PCDATA:** TagExpr. Specifies the code comprising the Live tag expression.

THE "SYNTHESIZE" ELEMENT

The "synthesize" element defines a Synthesize Script for the dossier, which properly generates the text to be output as the dossier. If multiple synthesize scripts are defined, their generated results are appended into one final text stream for output. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the Synthesize script.

THE "DOSSIER_EXPORT" ELEMENT

The "dossier_export" element defines the logic used in generating formatted data for use within other products, such as mapping tools and virtual table tops. The complete list of attributes for this element is below.

- **filename:** (Optional) Text. Specifies the default filename into which the generated data is output. The user is free to specify any filename he wishes, but this allows HL to automatically name the file correctly for programs with pre-defined import file mechanisms. If empty, the user is required to specify a filename. Default: Empty.
- **location:** (Optional) Text. Specifies the default location (i.e. file system path) in which the generated data is output. As with the "filename" attribute above, you can automatically output the file to a pre-determined location where the target program is expecting to find the file to import. If empty, the user is required to specify the location. Default: Empty.

The "dossier_export" element also possesses child elements that describe its contents. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

- **synthesize:** Zero or more "synthesize" elements may appear as defined by the given link. This element defines the Synthesize Script used to generate the text to be output.

THE "SYNTHESIZE" ELEMENT

The "synthesize" element defines a Synthesize Script for the dossier, which properly generates the text to be output as the dossier. If multiple synthesize scripts are defined, their generated results are appended into one final text stream for output. The complete list of attributes for this element is below.

- **PCDATA:** Script. Specifies the code comprising the Synthesize script.

EXAMPLE

The following example demonstrates what a "dossier" element might look like. All default values are assumed for optional attributes.

```
<dossier id="standard" name="Standard Character Sheet">
  <dossier_sheet grouping="AtTop" default="yes">
    <sheetref sheet="standard1"/>
    <sheetref sheet="standard2"/>
    </dossier_sheet>
  </dossier>

<dossier id="statblock" name="Character Statblock">
  <dossier_text styles="plain+html+bbcode" grouping="statblock">
    <synthesize>
      ~script code goes here
      </synthesize>
    </dossier_text>
  </dossier>
```

When the "standard" dossier is being rendered, the global tag "dossier.standard" is added to the portfolio. When the "statblock" dossier is being synthesized, the global tag "dossier.statblock" is added to the portfolio.

HIDDEN ELEMENT (DATA)

THE "HIDDEN" ELEMENT

The "hidden" mechanism comes into play when users want to modify existing data files while keeping their changes separate. This comes in extremely handy if there are certain standard things (e.g. skills, feats, etc.) in your campaign that are forbidden by the GM. The "hidden" mechanism flags an object so that HL treats the thing as if it does not exist, even though it remains technically present. This means a hidden object will never be shown to the user for selection. Each hidden thing is defined through the use of a "hidden" element. The complete list of attributes for this element is below.

- **type:** (Optional) Set. Designates the type of object to be hidden. Must be one of these values:
  - **thing:** The object must be defined via a "thing" element.
  - **Default:** "thing".

  NOTE! At the present time, only things can be hidden. The design of this mechanism allows it to be extended in the future if there is a need to hide other objects.

- **id:** Id. Specifies the unique id of the object to be hidden. An object of the specified type must exist within this unique id.

NOTE! Since hidden objects are treated as if they don't exist, any references to hidden objects will fail to resolve successfully. This may have nasty implications, because it means that a hidden thing that is bootstrapped by another thing will cause the second thing to fail to compile. Normally, this will simply create a cascading list of a few things that must all be marked as hidden, but sometimes it can cause circular dependency issues. When that occurs, use the "replace" mechanism on things to substitute a thing that is innocuous.

EXAMPLE

The following example demonstrates what a "hidden" element might look like. All default values are assumed for optional attributes.

```
<hidden type="thing" id="thingid"/>
```

EDIT THING ELEMENT (DATA)

THE "EDITTHING" ELEMENT

The purpose of the "editthing" element is to allow authors to define how new things can be created by users via the integrated Editor within HL. For every type of thing that users are allowed to create, a separate "editthing" element is defined.

[TBD] Details of this mechanism will be documented in the future.

FAQ ELEMENT (DATA)

THE "FAQ" ELEMENT

The data files for every game system provide the means for an author to develop and maintain a list of frequently asked questions (or FAQ). The FAQ serves as an organized repository of notes, explanations, tips, and whatnot for anyone using the data files. The FAQ is not intended as an actual user manual for the data files, but augments the user manual with bits of useful information that don't make sense to include directly within the manual. Each individual FAQ entry is defined through the use of a "faq" element. The complete list of attributes for this element is below.

- **id:** Id. Specifies the unique id of the FAQ entry. This id is used in all references to the FAQ entry.
- **order:** (Optional) Integer. Specifies the order in which this FAQ entry is to appear in the generated FAQ file. All FAQ entries are sorted based on this attribute, with the lower values appearing before higher values. If two or more FAQ entries have the same order value, they are sorted alphabetically based on the topic. This allows you to control the order in which FAQ entries appear, enabling organization of the contents. Default: "100".
- **topic:** Text. Specifies the "title" to displayed for this FAQ entry. A summary of all topics is generated at the top of the FAQ file, allowing users to see all the topics and click on those of interest to get direct access to the details.
- **PCDATA:** Text. Specifies the detailed description text to be displayed for this FAQ entry. The description may contain HTML tags, although care should be taken to ensure that the HTML used is very simple, else formatting problems can arise in the generated FAQ file.

EXAMPLE

The following example demonstrates what a "faq" element might look like. All default values are assumed for optional attributes.

```
<faq id="faqentry" order="1000"
   topic="Add your own FAQ entry here">
    Description of the FAQ entry
  </faq>
```

Authoring Examples

The goal of this section is to provide concrete examples of how to create data files for whatever game system you choose. Although additional examples are included, there are two primary components of this section. The first is an examination of the Sample data files provided by the Kit, which also serve as the starting point for creating data files for your own game system. The second is a detailed walk-through that guides you through the evolution of fully operational data files for the Savage Worlds game system. Starting with the Skeleton data files, you'll witness how the entire process unfolds.

What We Assume

For all of the examples below, we assume that you've familiarized yourself with the rest of the Kit documentation and all of the core concepts. Regular references will be made to the terms, concepts, and mechanisms that are introduced by the Kit. At a minimum, you must be comfortable with the material presented in the Basic Concepts and Terminology section.

If you wish to actually modify any of the data files, you will need a text editor that allows you to edit XML files. If you are only viewing the files, more web browsers provide a reasonable solution for displaying the contents of XML files.

For editing, you can get by with the Notepad editor included with Windows. However, that editor is extremely limited in what it offers, so we recommend you use something with a little bit more functionality if at all possible. Numerous text editors are available for free online that will work well, while commercial tools are also available. A few good options are listed below:

- **Notepad++:** [1] (http://notepad-plus.sourceforge.net)
- **EditPad Lite:** [2] (http://www.editpadpro.com/editpadlite.html)
- **Textpad.com:** [3] (http://www.textpad.com)
- **XMLMarker:** [4] (http://www.symbolclick.com)

Sample/Skeleton Data Files

To simplify the creation of custom data files for a new game system, the Kit provides the Skeleton data files as a robust starting point. The Authoring Kit Sample game system provides a working example that builds directly on top of the Skeleton data files. The major facets of the Skeleton data files are discussed in their own section, which can be accessed via the link below.

Savage Worlds Walk-Through

As mentioned above, the Kit includes complete data files for the Savage Worlds game system from Great White Games (http://www.greatwhitegames.com) . We've written up a detailed walk-through of how we created these data files, using the Skeleton files as our starting point. This should provide an excellent guide for how to adapt the Skeleton files to whatever game system you have in mind. Click on the link below to follow the process from start to finish.

Miscellaneous Examples

[TBD]

Skeleton Data Files

The Authoring Kit offers the Skeleton data files as framework that can be adapted to whatever game system you're interested in. They provide a viable starting point to build upon. The topics below delve into the various facets of the Skeleton data files. Concrete examples of how to use and adapt each facet can be found within the separate Savage Worlds Walk-Through.

- Data File Organization
- Definition File
- Control Elements
- Tags and Tag Groups
- Reusable Structural Blocks
- Actor Behaviors
- Adjustments
- Gear and Equipment
- Trackers and Resource Management
- Centralized Object Information
- Styles and Visual Resources
- Reusable Visual Blocks
- Tab Panels
- Summary Panels
- Forms
- User-Manipulable Things
- Behind-the-Scenes Things
- Starting Objects
- Character Output
- Transaction Handling
- Formalized Advancement
- Dashboard and Tactical Console

Savage Worlds Walk-Through

This section of the documentation outlines the evolutionary process of the Savage Worlds data files. It provides a technical walk-through of how the Skeleton data files provided with the Kit were converted into the fully functional data files for Savage Worlds. The process is outlined in a step-wise fashion that you can follow along with if you wish or simply use as a reference. For steps that involve the entry of numerous data elements (e.g. attributes, skills, edges, etc.), a few key examples are outlined and the rest remain as an exercise if you wish. Alternately, you can simply review them within the completed Savage Worlds files that have everything already implemented for you.

IMPORTANT! The Savage Worlds data files implement the rules as outlined in the Explorers Edition rulebook. In some cases, pertinent mechanics that existed in the original rulebook are also included to illustrate how to do various tasks.

What We Assume

This walk-through assumes you have a basic familiarity with the Savage Worlds game system from Great White Games, since our goal is to develop data files for that game system. If you aren't familiar with the game, the first step is to get up to speed on the

basics. This can be easily accomplished by reviewing the free "Test Drive" rules for Savage Worlds, which can be found online. Just go to the Great White Games website (http://www.greatwhitegames.com) and you should find the file in the Downloads section. If you have trouble locating the file, you can use this mirror link (http://www.lonewolfdevel.com/downloads/savage-worlds-test-drive.pdf).

If you have not done so yet, you will likely benefit by reviewing the organization and content of the Skeleton/Sample data files. Since the Savage Worlds data files start with the Skeleton data files and evolve from there, you'll probably find that extra context to be helpful.

Lastly, the assumptions sets forth for the examples as a whole also apply.

Where to Find the Source

Sometimes, it might be useful to refer to the source code of the completed Savage Worlds files during this tutorial. You can find it by downloading and importing the Savage Worlds game system from the updates mechanism (click "Locate Files" on the "Select Game System" form that appears when you start Hero Lab), then looking in the "source" directory under the Savage Worlds folder. Depending on where you installed Hero Lab, this will be something like:

```
c:\herolab\data\savage\source\
```

Once there, you can view all the source files. Note that editing them will NOT apply any changes to the Savage Worlds game system - they are for reference only. If you want to make changes to the game system, you must copy them to a different folder (for example c:\herolab\data\mysavage) and begin editing there.

Creating the Savage Worlds Data Files

The topics below follow the evolutionary process from start to finish. If this is your first time reading through this documentation, we strongly recommend that you read through these topics in the order presented, as each one builds upon changes made in the previous stages. After an initial reading, you can return to the topics that most closely pertain to whatever task you are working on within your own data files.

- Getting Started
  - The First Steps
  - Develop Your Plan
  - Initial Changes
- Fundamental Pieces
  - Traits
  - Attributes
  - Skills
  - Derived Traits
  - Resources
- Core Game System Elements
  - Basics of Arcane Backgrounds
  - Races and Racial Abilities
  - Verifying Actor Pre-Requisites
  - Encumbrance
  - Hindrance Support
  - Adding Hindrances
  - Countering Hindrances
  - Edge Support
  - Adding Edges
- Adding and Revising Initial Panels
  - Revise Configuration Form
  - Prune Extraneous Material
  - A Tab for Skills
  - Revise "Basics" Tab
  - Incrementer Behavior
  - Add an "Edges" Tab
  - Evolving the "Edges" Tab
- Evolving Game System Elements
  - Character Creation Logic
  - Revise Adjustments
  - More on Arcane Backgrounds
  - Add an "Arcane" Tab
  - Refining Arcane Backgrounds and Powers
  - Synthesizing Descriptions for Display
  - Refining the "Arcane" Tab
  - Damage
  - Revamp the "Static" Form
- Character Advancement
  - Advancement Support
  - Adding Advancements
  - Revise Journal Tab
- Gear and Equipment
  - Equipment
  - Weapons
  - Ranged Weapons
  - Finishing Off Weapons
  - Armor
  - Gear and Load Limit
  - Natural Weapons
  - Special Weapons
- Expanding Our Coverage
  - Conferring Edges and Hindrances
  - Injuries
  - Fright
- Vehicles
  - Basic Vehicles
  - Complex Vehicles
- Refinement of Behaviors
  - Using Bitmaps for Die Types
  - Show the Derivation of Values
  - Revise Summary Panels
- Character Output
  - Character Sheet Design
  - Character Sheet Output
  - Implementing the Character Sheet
  - Character Sheet Phase 2
  - Character Sheet Phase 3
  - Character Sheet Phase 4
  - Character Sheet Phase 5
  - Character Sheet Refinement
  - Statblock Output
- Assorted Remaining Elements
  - Miscellaneous Cleanup
  - Weird Science Gizmos
  - Dashboard
  - Tactical Console
  - Initiative
  - Allies
  - Character Sheet Additions

    - User-Configurable Options
    - Specialized Edges
- Other Character Types (NPCs and Creatures)
    - NPC Support
    - More Cleanup
    - Creatures
    - Creature Specification
    - Creature Refinement
    - Creature Customization
- The Final Stages
    - Final Cleanup
    - Final Cleanup Continued
    - Integrated Editor Support
    - Stock Portfolios
    - Data File Release

CHANGES AFTER INITIAL RELEASE

The Savage Worlds data files are maintained on an ongoing basis, which means bugs are fixed and new capabilities are sometimes added. Instead of retrofitting the preceding documentation, any such changes are outlined in the following topics.

- Changes in V3.2

GETTING STARTED

THE FIRST STEPS (SAVAGE)

Before you embark on this process, there are a number of important features of HL that you should be familiar with. The sections below outline these features.

ENABLE DATA FILE DEBUGGING

Begin by making sure that you've configured HL to enable the debugging aids. By default, HL assumes that users are not editing their own data files, so various development facilities within the product are disabled. You need to make sure they are turned on. Go to the "Develop" menu, where you need to turn on the "Enable Data File Debugging" option.

COMPILING DATA FILES

During the course of developing your data files, there will be times where you want to fully test that everything is working properly. There will also be times when you simply want to verify that your changes are syntactically valid and compile successfully. You can ask HL to re-compile your data files at any time by going to the "Develop" menu, where you can invoke the "Compile Data Files" option. You'll be prompted to specify the game system to re-compile, and you'll be shown any error messages that might be encountered during the compilation process.

As long as your files fail to compile, they will not load into HL, so you should get in the habit of frequently re-compiling your data files. This will uncover problems quickly, since the error must exist in whatever changes you've made since the previous successful compile.

As a shortcut, you can use the <Ctrl-C> key combination to invoke a compile. This makes it easy to regularly verify that your data files are valid at each step along the way as you develop them. Please note that the <Ctrl-C> key combination will not work when the input focus is a text edit portal (e.g. character name), since <Ctrl-C> is interpreted as a traditional "Copy" command within a text edit portal.

USING QUICK-RELOAD

Whenever you make changes to your data files, you need to load those changes into Hero Lab so you can use and test them. The simplest way to do this is to go to the "File" menu and select "Switch Game System". However, this approach always shows you the release notes for the game system and potentially the "demo mode" warning, after which you'll be shown the "Configure" form for a new character. After a few dozen times, this gets really old.

So HL includes the "Quick Reload" mechanism, which can be invoked by going to the "Develop" menu, where you can select the "Quick Reload" option. This mechanism re-compiles the data files, if necessary, and then reloads them into HL, bypassing the extra steps and restoring the current tab that is selected. As an added bonus, if you have a saved portfolio loaded, your portfolio is reloaded. This makes it quick and easy to incrementally modify and test out behaviors associated with selected options.

A keyboard shortcut is also provided for the quick-reload mechanism. When you use the <Ctrl-R> key combination, HL will trigger a reload. When all you are doing is structural changes, re-compiling is a good check that everything still works, but using the quick- reload mechanism will often be a better technique. With the quick-reload, you get to visually witness the impact of your changes, so may prefer to use quick-reload on a regular basic during development instead of just re-compiling.

TAKE SNAPSHOTS REGULARLY

As you evolve your data files, you will be making significant changes. If you aren't extremely careful, it's likely that you will end up causing everything to break at a few points along the way. When this happens, it can be invaluable to be able to see exactly what has been changed since the last time everything was working fine. In order to do this, you need to have a saved copy of when things were last working. So we strongly recommend that you take snapshots of your data files at regular intervals, preferably at milestones where everything is working the way you want it.

The easiest way to take a snapshot is to use the "HLExport" tool that is included with HL. This tool is designed to package up all of the data files for a game system into a single file that can be imported back into HL. Using HLExport, you can readily take snapshots of your working data files and save them. If you need to refer back to one, you can import the file back into HL. However, be sure to import the files into a different directory from your data files, else the imported files will overwrite your recent changes.

Another simple technique is to make a copy of the entire directory contents for your game system. This allows you to do a direct file-to-file comparison of any file at any time, which can be quite handy at times. In fact, a combination of these two methods will often provide the best (and safest) results.

KEEP A NOTEBOOK HANDY

Throughout the entire process of data file development, there will be lots of details you think of that you can't implement immediately. But you certainly don't want to forget any of these details. We highly recommend that you keep a notebook readily available during the evolution of your data files. Jot down notes about changes or potential problems that you identify along the way, then keep the notes as a running "to do" list. At various junctures, re-check your list and address whatever tasks make the most sense at the time.

DEVELOP YOUR PLAN (SAVAGE)

Before writing any data files, it's critical that you first do your homework. If you launch into writing the data files without a good plan, you will likely run into a number of surprises and setbacks along the way. While such events won't stop you from successfully creating the data files, they will almost certainly cause unnecessary delays and frustration. So you will fare best if you take the time upfront to develop your basic implementation strategy, map out all of the basic building blocks you'll need, and design how everything should look and behave for the user.

IMPLEMENTATION STRATEGY

The first thing you need is to figure out how you are going to go about developing your data files. We've provided you with a solid starting point in the Skeleton data files, and now those files need to be adapted. The sequence used for the Savage Worlds data files leverages a methodology that has worked well for a number of game systems. You are welcome to come up with your own approach, as there is no "right" way to do it, although we recommend you start with the methodology we've presented and adapt it to suit your needs.

Building Blocks Take the time to look closely at the game system for which you are developing data files. There are probably lots of subtle facets of the rules mechanics that you take for granted, but that will introduce complexities in writing the data files. In some cases, you will uncover holes in the wording of the rules that the game designers didn't intend. When you embark on writing data files for a game system, you have to tell HL exactly how things work, with none of the grey areas that the human brain adapts to so well.

Put together an outline of all the various building blocks for the game system. Map these pieces to the corresponding mechanism for representing each piece via the Kit. Also look at the various supplement books for the game system to get a concrete grasp of the new wrinkles introduced in those books, then figure out how those will impact your initial outline. When you're done, you should have a detailed outline of the various game system elements and a good idea of how you're going to manage them within HL.

The following is a list of some of the key Kit mechanisms to focus on during this process:

- components and fields
- component sets
- tag groups and tags
- phase and evaluation cycle
- entities

LOOK AND FEEL

Now take some time to evaluate how everything will behave for the user. The overall organization of the interface is critical to highly usable data files, such as what gets presented, how and where it gets presented, and how the user is allowed to make appropriate selections. Also give some thought to how you'll provide feedback to the user about the characters being created and any errors that might arise.

Once you have you have your visual organization and layout mapped out, you can translate that into the mechanisms provided by the Kit. This includes assessing where to use tables, choosers, menus, etc. It also entails determining what the contents of tables will look like and what portals the user will interact with.

We highly recommend that you sketch out how everything will look on paper. Sketches of each tab and how the information will be organized on each are invaluable. For some tables, additional sketches of the contents of each item in the table are also extremely helpful.

VISUAL SKINS

The last planning step is how you intend to visually "skin" the interface. The skin is the set of colors, fonts, textures, icons, and bitmaps that comprise how everything looks to the user. We highly recommend you defer any work on the skin until after your data files are otherwise completed. Once you have all the core behaviors operating correctly, you can switch your attention to the skin details. That being said, though, it is always a good idea to give a little thought to how everything will look at the outset, just in case there are layout implications due to your intended final look that need to be considered along the way.

INITIAL CHANGES (SAVAGE)

When you finally get underway creating your data files, there are a few things that you must always do first, as outlined in the topics below.

CREATE THE GAME SYSTEM

The very first step is to create your new game system. HL can automatically setup a new game system for you, starting you out with a set of data files that you can adapt and build upon. Go to the "Develop" menu and select the "Create New Game System" option. For the Savage Worlds data files, we'll specify the name of the game system as "Savage Worlds" and the folder to use as "savage". After a few moments, HL has set everything up properly, so you can go to the "File" menu, select "Switch Game System", and enter your new game system.

REVIEW THE SAMPLE/SKELETON DATA FILES

The framework for your new game system is now setup. Since this framework provides all of the basics you'll need and must simply be fleshed out, we refer to this starting point as the "Skeleton" data files. If you haven't done so yet, now would be an ideal time to review the Kit documentation regarding the Skeleton data files, as it provides suggestions for how best to evolve them into a full-fledged game system.

You should also take the time to review the actual contents of the data files themselves and get familiar with them, as you'll begin modifying and adapting them in just a moment.

UPDATE GAME SYSTEM DETAILS

Once the new game system is in place, the first change to make is to modify the definition file for your game system. Open the file "definition.def" for editing. Near the top, you'll find the "game" element with assorted attributes containing placeholder information about the game system. Modify these attributes to accurately reflect the details for Savage Worlds. The result should be something like below.

```
<game
  game="Savage Worlds"
  publisher="Great White Games"
  website="http://www.greatwhitegames.com/"
  copyright="Copyright © 2004 by Great White Games. All
rights reserved."
```

```
    legaloutput="Copyright © 2004 by Great White Games. All
rights reserved."
    manualroot="savagemanual.htm"
    editorroot="authoring\savageedit.htm">
    </game>
```

Beneath the "game" element, you'll find the "author" element. Update the attributes for this element with appropriate details about you, the author. You should end up with something that parallels the example below.

```
<author
    author="Lone wolf Development"
    email="info@wolflair.com"
    website="www.wolflair.com">
    </author>
```

Once these changes are made, use the quick-reload mechanism by pressing <Ctrl-R>. This re-compiles and re-loads the data files to make sure everything is done correctly.

RELEASE INFORMATION

Next up, you should revise the "release" element in the file "definition.def". The large block of description text at the end of the element is displayed to the user whenever the data files are loaded. This text should be revised to point the user in the right direction when first loading the data files. It should provide an overview of the data files, where to look for help, and anything else you think is important. You should also include a suitable legal notice regarding the data files and the game system. The final material for the Savage Worlds game system is shown below.

```
<release
    major="1"
    minor="0"
    required="3.1"
    summary="Core data files for Savage
Worlds"><![CDATA[{b}{text ffff80}Savage Worlds data files   for
Hero Lab{text 010101}{/b}

 Welcome to the Savage Worlds data files for Hero Lab! For
details on using Hero Lab, please refer   to the Tutorial and/or
User Manual documentation that will be found with Hero Lab
under the Start   menu. A separate manual with specifics for the
Savage Worlds data files can be accessed via the   "Help" menu
and the "Configure Hero" form.

 These data files represent a complete implementation of the
Explorers Edition rules for the Savage   Worlds RPG from Great
White Games. You can create any type of character outlined in
the rulebook   using these data files. Many items have mouse-
over information associated with them, so if you   see an icon
that isn't familiar, be sure to mouse over it for more information.

 If you have questions or comments about Hero Lab, our public
discussion forums can be found at   the following URL:
{u}~[(@http://support.wolflair.com/index.php?
name=MDForum&file=index]http://support.wolflair.com/~[(#]{/u}
There are separate forums for the   product in general, each
game system, and using the Authoring Kit. Please post your
thoughts in   the appropriate forum so that it is seen by the right
people and responded to accordingly.

 These data files are published under license from Great White
Games.

 Happy gaming!

 {b}{text ffff80}Legal Information:{text 010101}{/b}

 Copyright © 2008-2009 Great White Games. All rights
reserved. Reproduction without the written   permission of the
publisher is expressly forbidden, except for the purposes of
 ]]>
   </release>
```

IMPORTANT! Always make sure that you specify the "required" attribute appropriately. This should typically be the current version of Hero Lab at the time you release your data files. Doing so ensures that the data files can only be loaded into a version of HL that fully supports the data files you've created.

FUNDAMENTAL PIECES

TRAITS (SAVAGE)

When converting the Skeleton data files to the new game system, the first thing to address is the traits. So that's where we'll begin with the Savage Worlds data files.

WHAT ARE TRAITS?

Virtually every game system utilizes an assortment of game mechanics to represent facets of a character. For example, there are attributes, skills, abilities, and a host of other possibilities. Each of these has one thing in common: a single rating. Any facet of a character that includes a single measurement or rating is considered to be a "trait".

Since the nature of components allows extensive reuse of fields and behaviors, the notion of traits is important. All of the shared behaviors of traits can be encapsulated within a single, reusable component. As a result, the Skeleton files define a shared "Trait" component. Each different type of trait has its own component that manages only the facets that are different for that particular trait. This approach makes data file development significantly easier, because you don't have to worry about re-implementing the same basic logic multiple times for attributes, skills, abilities, and whatever else is employed by the game system.

GENERAL TRAITS VERSUS SPECIFIC TRAITS

In many game systems, there are multiple facets of the game mechanics that share common behaviors, and there are some facets that are distinct. When looking at traits in general (attributes, skills, abilities, etc.), the question of commonality is critical. The Skeleton data files define a "Trait" component that embodies these common behaviors, and there are other components for the differences. So the game system must be assessed to determine which behaviors are shared and which are distinct. Shared behaviors should be modeled within the "Trait" component and the others in the appropriate, separate components.

The first thing to do is review the game system to identify the shared behaviors that apply to multiple traits. These behaviors can then be folded into the "Trait" component for general use.

DIE TYPES FOR TRAITS

The most obvious facet of traits in Savage Worlds is the use of different die types instead of raw values. HL doesn't have any built-in support for something like this, so we need to improvise a solution. Fortunately, it's an easy task. The various die types use by Savage Worlds are a d4 through a d12, in increments of two. That translates to a range of 2-6, in increments of one, with the final value multiplied by two.

This can be easily implemented by setting up traits to have minimum value of two and a maximum value of six, with a script to convert the value displayed to show the proper die type instead. We can modify the data files to support this. Looking at the data files, the "Attribute" component has a script that sets the minimum value to one. We can move this script to the "Trait" component so it applies to all traits, and we can modify it to assign the appropriate minimum and maximum values. The resulting script should look something like the one below:

```
<eval index="2" phase="Initialize" priority="3000"><![CDATA[
  ~since die types range from d4 to d12, in multiples of 2, we
have a range of
  ~2-6 for traits - we'll convert to the die type for display in a
separate script
  field[trtMinimum].value = 2
  field[trtMaximum].value = 6
  ]]></eval>
```

Once the value is tracked correctly, we need to convert it properly for display. We plan on using an incrementer to allow the user to adjust attributes and skills, just like is done for other game systems. Incrementers rely on the finalized value of the corresponding "user" field to display the proper value. Consequently, we need to ensure that the Finalize script for the "trtUser" field retrieves the correct text to display. However, we may also want to display the generated text in other situations, such as within character sheets and other places in the interface. The solution is to define a new field where we'll synthesize the text for display - we'll call this field "trtDisplay". The "trtUser" field can then grab the contents of this new field within its Finalize script. The field should look like below.

```
<field
  id="trtDisplay"
  name="Net Roll to Display"
  type="derived"
  maxlength="50">
  </field>
```

The text for the "trtDisplay" field needs to be generated prior to its use in the Finalize script of the "trtUser" field. The problem is that a field with a Finalize script cannot use the finalized value of a different field. The solution is to use a script that is timed to occur very late in the evaluation process. Since we can only compute value-based fields in a Calculate script, and we need to synthesize a text- based field, we must generate the text in an Eval script. Our script gets the final value for the trait and multiplies it by two to yield the proper die type, which can be output with an appropriate notation (e.g. "d6"). After our data files get more fleshed out, we can replace the simple "d6" notation with a suitable bitmap for the die type.

There are two drawbacks with this approach that we must handle. First of all, in-play adjustments can potentially increase or decrease a trait beyond the standard limits. Consequently, we must bound the value appropriately before we attempt to convert it. Secondly, all traits are now shown using the die type syntax, but traits include derived traits, and derived traits are simple values and not die types. As such, we need to ensure that derived traits display as the actual value and do not get converted. This can be achieved by amending the script to handle derived traits specially. Since every derived trait inherits from the "Derived" component, we can identify a derived trait based on the "component.Derived" tag and handle it specially.

The resulting script looks something like the one below:

```
 <eval index="3" phase="Render" priority="5000" name="Calc
trtDisplay">
   <after name="Calc trtFinal"/><![CDATA[
   ~if this is a derived trait, our display text is the final value
   if (tagis[component.Derived] <> 0) then
    field[trtDisplay].text = field[trtFinal].value
    done
    endif

  ~bound our final value including in-play adjustments
   var final as number
   final = field[trtFinal].value
   if (final < 2) then
    final = 2
   elseif (final > 6) then
    final = 6
    endif

  ~convert the final value for the trait to the proper die type for
display
   var dietype as number
   var display as string
   dietype = final * 2
   display = "d" & dietype

  ~put the final result into the proper field
   field[trtDisplay].text = display
   ]]></eval>
```

Once everything is in place, reload the data files and test everything out. The sample attribute should display as "d4" on the Basics tab. If you adjust it up and down, it should properly display the next die type, capping at "d12" at the upper end and "d4" at the lower end.

NOTE! Instead of relying on a range of 2-6 and then multiplying by two, we could use the actual die type (4, 6, 8, etc.) and ensure that all adjustments apply a delta of two. For example, we could define all incrementers for editing attributes and skills to have an "interval" of two. Either approach is valid. However, in Savage Worlds, the "half" value of the die type is often used, so we'll either be adding/subtracting/dividing by two everywhere or simply multiplying by two before displaying the results. It's easier to just multiply by two immediately before showing the info to the user and simply adjusting by one step everywhere, so that's the approach we've chosen.

BONUSES AND PENALTIES

There's another important facet of traits in Savage Worlds that needs to be addressed. Traits can have bonuses and/or penalties applied to the actual roll - which is distinct from adjustments to the die type. For example, a character might have a trait roll of "d6+1" due to a suitable bonus. These adjustments need to be both tracked and displayed appropriately.

Tracking the adjustments can be readily accomplished by adding a new field. Since the adjustment applies to the actual roll, we'll call the field "trtRoll". Scripts can modify this field, which defaults to zero, to confer bonuses or penalties to rolls for a given trait.

But wait. The "Profession" edges within Savage Worlds work differently than normal adjustments to traits. By default, adjustments to rolls will "stack", meaning that a "+1" from one source combines with a "+1" from another source to yield a net adjustment of "+2". However, the "Profession" edges are specifically defined to not stack with each other - but they do stack with other adjustments. Since adjustments that do stack can be

combined with adjustments that don't stack, we need to track the two independently. This means that we actually need to add one more field to track the non-stacking "Profession" adjustments. To keep everything clear, we'll call this field "trtNoStack".

Now we need to display the adjustments properly. This entails factoring the adjustment values (there are two) into the text that is displayed for the trait. So we need to add the code below to the Eval script we created above to tack on the net adjustment value. This extra logic should be inserted after the die type is synthesized and before the final value is copied into the field.

```
~if there are any bonuses or penalties on the roll, append those
to the final result
  var bonus as number
  bonus = field[trtRoll].value + field[trtNoStack].value
  if (bonus > 0) then
    display &= "+" & bonus
  elseif (bonus < 0) then
    display &= bonus
  endif
```

Scripts are going to need to access our new fields to get and set the values. Since adjustments are pretty common in Savage Worlds, we're going to need to do this in lots of places. To make this easier throughout our scripts, we can define script macros that provide us with a shorthand for accessing the fields. Just below the "behavior" element within the file "definition.def", you'll find a small number of macros already defined. We'll add two new macros that correspond to our new fields, adding the two XML elements shown below.

```
<scriptmacro name="traitroll" param1="trait"
    result="hero.childfound[#trait].field[trtRoll].value"/>
<scriptmacro name="traitprof" param1="trait"
    result="hero.childfound[#trait].field[trtNoStack].value"/>
```

SWITCHING THE DISPLAY

The Skeleton data files are centered around the "trtFinal" field of the "Trait" component, but we've just changed everything so that the "trtDisplay" field actually contains what we want to show to the user. Consequently, we should go through the data files and assess all the references to "trtFinal" to determine which of them should be converted over to the "trtDisplay" field. Wherever we need to utilize the value, "trtFinal" is still the proper thing to use, but we should use "trtDisplay" anywhere that the final result should be displayed to the user.

Doing a search of all the data files, we find quite a few references to the "trtFinal" field. We'll examine each and convert as appropriate. Remember that the "trtDisplay" field is a "text" field, so we need to use "field[trtDisplay].text" to access it.

In the file "definition.def", both references are within script macros, and those macros need to specifically manipulate the "trtFinal" field value. So no changes are needed here.

In the file "traits.str", there are multiple references, but they are all part of the proper handling of traits that we just implemented above, so nothing needs to be changed.

In the file "tab_basics.dat", there is one reference. It's in a table that displays traits as picks. Since we're showing picks - not things - we should convert to the new field. So we change the reference to the line of script code shown below.

```
@text = field[trtDisplay].text
```

In the file "summ_basics.dat", there is one reference. Since this is a summary panel, it's definitely a situation where we want to display the final text to the user. So the field associated with the portal must be changed from "trtFinal" to "trtDisplay".

In the file "form_taccon.dat", there are two references. Both are used to display traits in generic fashion, so both must be converted.

In the file "procedures.dat", there are four references. The first is used for displaying traits in general ("dshrolls" procedure), so it must be converted. The next is found in the "dshcombat" procedure and must be converted for the same reason. The remaining references are in the "dshbasics" procedure, where the first is for power points and must remain a value (i.e. no conversion), while the other two are for display of traits like attributes, so they can be converted.

In the file "sheet_standard1.dat", there are five references. Only two of the references pertain to attributes and skills, with the other applying to traits where a numeric value is needed. So we only change the references to "trtDisplay" in the templates "oAttrPick" and "oSkillPick".

In the file "out_statblock.dat", there are two references. Both are used for the display of traits, so both can be converted. The second instance is specifically for derived traits, so the conversion is unnecessary, but it is still safe (and well-advised) to make the change.

ATTRIBUTES (SAVAGE)

Every game system has a core set of attributes, although they are sometimes referred to using a different term (e.g. stats, characteristics, etc.). These attributes are the cornerstone of the game system and everything builds upon them, so they are the natural next step.

HOW TO HANDLE ATTRIBUTES?

Looking at how Savage Worlds works, there are five core attributes to the game, and we've already made a number of changes that apply to traits in general. But is there anything else of interest the needs to be handled specifically for attributes? Nothing else is apparent, so we don't need to make any changes to the "Attribute" component at this point.

ADD ATTRIBUTES FOR GAME SYSTEM

Once we're confident that we have all the necessary mechanics in place for managing attributes properly, we need to define them. Open up the file "thing_attributes.dat" and we'll start adding each new attribute. [Note: For the moment, do not delete the sample attribute.]

Since all of the mechanics for handling attributes is orchestrated by the underlying components, the definition of each attribute is incredibly simple. You can copy the sample attribute provided and adapt it appropriately for each attribute of the game system. Be sure to specify the "trtAbbrev" field with an abbreviation for the attribute. Also, be sure to assign a suitable "explicit" tag to each attribute, with the value corresponding to the order in which the attributes are to be shown to the user. The "Agility" attribute is shown as an example below.

```
<thing    id="attrAgi"
   name="Agility"
   compset="Attribute"
   isunique="yes"
   description="Agility is your hero's nimbleness, quickness, and
dexterity.">
   <fieldval field="trtAbbrev" value="Agi"/>
   <tag group="explicit" tag="1"/>
   </thing>
```

Once you've added all the new attributes, reload the data files and verify that they all show up properly.

DELETE SAMPLE ATTRIBUTE

Now that our attributes appear to be working properly, we can delete the sample attribute. Unfortunately, this will cause our data files to cease compiling successfully. The reason is that the sample attribute is actually referenced in places within the Skeleton data files (and now our game system). So we need to locate those places and correct them.

The only place of importance where the sample attribute is referenced happens to be as the linked attribute for the skills given in "thing_skills.dat". Open this file and locate references to "attrSam" - there should be three of them. We're going to be re-working skills in a moment, so we simply need a temporary fix to make the compiler happy. Change each of the skills to reference a different attribute instead, such as "attrAgi".

At this point, you should be able to compile and reload the data files. The five attributes for Savage Worlds are now in place and behaving as they should.

SKILLS (SAVAGE)

Once attributes are working, we need to focus on the next most fundamental facet of the game system, working our way progressively so that we always build on top of the mechanisms we've implemented thus far. For Savage Worlds, the next facet is skills.

HOW TO HANDLE SKILLS?

As we did for attributes, the first thing we need to do for skills is figure out what mechanics we haven't handled yet. After close examination, there are a number of aspects of skills in general that require special handling distinct from attributes. We also need to assess any differences between how skills are handled within the Skeleton files versus the Savage Worlds game system. Lastly, we need to review each individual skill to see if any of them will introduce complexities, handling them as appropriate.

AUTO-ADD OF SKILLS

The most obvious difference between skills in Savage Worlds and the Skeleton data files is that all skills are auto-added by the Skeleton files. We can stop auto-adding skills easily enough, but then we'll need to modify the tab panel to let the user add skills. That entails a bit more work that we aren't ready to do yet, so we'll defer fixing this issue for a little while.

CALCULATED ATTACK VALUES

Within the Skeleton files, attack values for weapons are calculated by adding a value for the weapon to the underlying skill and its linked attribute. Weapons work nothing like this for Savage Worlds. As such, the calculations of weapon attack values that are provided by default need to be changed. In addition, these calculations reference the "Melee" and "Shooting" skills that are pre-defined - and that are about to be replaced. We could try to figure out how this should work now, but it's premature to do so. The simplest solution is to just comment out the script code that calculates the attack values for now and replace it with something appropriate when it's time to look at adding weapons.

Open the file "equipment.str". If you search for references to the "wpNetAtk" field, you'll find its definition and two separate Eval scripts that calculate its value. One script is within the "WeapMelee" component and the other is in the "WeapRange" component. Comment out these scripts, but don't delete them - we'll replace them with proper logic later on. To comment them out, you can either place a "~" at the start of each line of script code, or you can put the entire XML "eval" element within an XML comment wrapper. Both methods will work equally well, and both are shown below.

```
<!—
 <eval index="1" phase="Final" priority="7000" name="Calc
wpNetAtk"><![CDATA[
   field[wpNetAtk].value = #trait[skShooting] +
field[wpBonus].value + field[wpPenalty].value
   ]]></eval>  -->

OR

<eval index="1" phase="Final" priority="7000" name="Calc
wpNetAtk"><![CDATA[
   ~field[wpNetAtk].value = #trait[skShooting] +
field[wpBonus].value + field[wpPenalty].value
   ]]></eval>
```

KNOWLEDGE SKILL

After reviewing all of the skills, only one of them has any interesting implications - the "Knowledge" skill. Within Savage Worlds, skills are typically unique, existing only once. However, the Knowledge skill must have an appropriate "domain" specified for it, and it can be taken multiple times with separate domains for each. In order to support this, a few changes need to be made to how skills work in general.

First of all, the "Skill" component set currently forces all skills to be designated as "unique". The "Knowledge" skill must not be a unique skill so that it can be purchased multiple times, while all other skills should remain unique. Therefore, the "forceunique" attribute within the "Skill" component set definition (near the bottom of the file "traits.str") needs to be eliminated. It's important that you **not** change the attribute to have a "no" value, since that will force all skills to be non-unique. Instead, we want the default behavior, which allows some skills to be designated as unique and others as non-unique.

The remaining facet of the "Knowledge" skill that needs to be handled is allowing the user to specify the appropriate "domain". This requires that we add a new text-based field to store the domain, as well as a way to generically identify skills that require the user to specify a domain. Since this is a mechanic that many game systems require, the Skeleton files include built-in mechanisms to handle it. The "Domain" component is provided such that you can simply integrate it into a component set to provide domain tracking, so add a new "compref" element to the "Skill" component set, as shown below.

```
<compset
  id="Skill">
  <compref component="Skill"/>
  <compref component="Trait"/>
  <compref component="Domain"/>
  <compref component="CanAdvance"/>
  </compset>
```

Similarly, the "User.NeedDomain" tag is defined as a convenient means to designate a thing that requires a domain, so we can assign this tag to the "Knowledge" skill. It is defined as a "User" tag because the tag will be able to be specified by users who create their own custom skills through the Editor.

ADD SKILLS FOR GAME SYSTEM

All of the necessary mechanics should now be in place for skills, so it's time to define them. Open up the file "thing_skills.dat" and you can begin adding all the Savage Worlds skills to the file. [Note: For the moment, do not delete the sample skill.]

As we discovered for attributes, all of the mechanics for skills are managed by the underlying components, so each skill is quite simple. You can copy the sample skill provided and adapt it appropriately for each skill of the game system. The "Investigation" skill is shown as an example below.

```
<thing
  id="skInvest"
  name="Investigation"
  compset="Skill"
  isunique="yes"
  description="Description goes here">
  <fieldval field="trtAbbrev" value="Invst"/>
  <tag group="DashTacCon" tag="NonCombat"/>
  <link linkage="attribute" thing="attrSma"/>
  </thing>
```

When defining skills, you'll need to specify the linked attribute for each. This is accomplished via the "link" XML attribute, as shown above. Every skill requires a linkage, and the compiler should report an error if you omit the linkage for a particular skill.

Any skill that should be shown within the non-combat list of skills on the Tactical Console needs to be flagged as such. You can do this by assigning the "DashTacCon.NonCombat" tag to the skill. When assigned, the skill will be included. We'll review this in more detail later on.

The one skill that will be slightly different from the others is the "Knowledge" skill. As mentioned previously, this skill must be non-unique and must have the "User.NeedDomain" tag assigned. The "Knowledge" skill is shown below as an example.

```
<thing
  id="skKnow"
  name="Knowledge"
  compset="Skill"
  description="Description goes here">
  <fieldval field="trtAbbrev" value="Know"/>
  <tag group="User" tag="NeedDomain"/>
  <tag group="DashTacCon" tag="NonCombat"/>
  <link linkage="attribute" thing="attrSma"/>
  </thing>
```

Once you've added all the new skills, reload the data files and verify that they all show up properly.

DELETE SAMPLE SKILL

Once the skills all show up and work properly, it's safe to delete the sample skill. Unlike with attributes, the sample skill is not referred to by anything else and can be freely deleted.

DERIVED TRAITS (SAVAGE)

The next thing to address in our conversion process is the set of derived traits for Savage Worlds. There are a small number of derived traits in the game, and each is interesting in a particular way, so each is described in the sections below. These traits should be added to the file "thing_traits.dat". [Note: For the moment, do not delete any of the existing derived traits.]

TRAITS IN GENERAL

Each derived trait uses its own Eval Script to calculate the derived trait value. Within these scripts, the "#trait" macro can be used to get the value of a trait, but the trait is assumed to exist for the character. In the case of skills that may not be added to a given character, the "#traitfound" macro is recommended instead, since a missing skill will result in a value of zero being used as a safe default.

Since derived traits are based upon other traits, we have a "chicken-and-egg" situation with respect to script evaluation timing. The "Trait" component auto-calculates the final value for the traits it derives from. But derived traits need to calculate themselves based on the final value of other traits. To solve this, the "Derived" component provides a second calculation of the trait's final value that occurs **after** the initial calculation, thereby allowing the final value of base traits to be utilized. This means that each derived trait must determine its own adjustments between the evaluation of these two scripts. We accomplish this by scheduling the Eval script for each derived trait to occur at Traits/4000, splicing it between the two component scripts.

The value of each derived trait is calculated as a "bonus" for the trait, using the "trtBonus" field. This field is used by the Eval script within the "Derived" component, just as it is used by the "Trait" component. We could introduce a new field for this purpose, but it's not necessary, provided we get our script timings correct.

In general, trait bonuses and penalties can arise from multiple sources. Consequently, we need to **add** our calculated value to the "trtBonus" field instead of setting the value. There are some situations where we need to explicitly set the value of a derived trait based on other picks and other situations where we need to add an adjustments based on other picks. The "Pace" trait is a perfect example of this, where certain races and edges set a new base trait value, while other effects can simply adjust the pace. This gets more complicated, and we'll discuss it in detail with the "Pace" trait below.

Appropriate tags must be assigned to each derived trait to control where and how the trait is displayed to the user, as well as the order in which the traits are displayed.

TOUGHNESS TRAIT

The "Toughness" trait is probably the simplest trait, so we'll start with it. Since the "Toughness" trait is needed both in and out of combat, we assign it tags to ensure it gets displayed in all the appropriate places. Other than that, the key item of interest for the "Toughness" trait is its calculation.

The rules stipulate that "Toughness" is calculated as "2 plus half your Vigor". In addition, the effects of any armor are also added. Since our "Vigor" attribute is already being tracked as a range of 2-6

and then doubled for display, our internal value is the "half" value that we need to add. We're going to defer handling any overage beyond a "d12" rating until later. To make our life easier, the Skeleton files pre-define a "#trait" macro for use in scripts that will readily access the value of a trait for us.

Putting it all together, the "Toughness" trait should look something like the "thing" element shown below:

```
<thing
  id="trTough"
  name="Toughness"
  compset="Trait"
  isunique="yes"
  description="Description goes here">
  <fieldval field="trtAbbrev" value="Tgh"/>
  <tag group="explicit" tag="9"/>
  <tag group="DashTacCon" tag="Combat"/>
  <eval index="1" phase="Traits" priority="4000">
    <before name="Derived trtFinal"/>
    <after name="Calc trtFinal"/><![CDATA[

    ~toughness is 2 plus half the character's Vigor, but we track
attributes at
    ~the half value (2-6), so we add Vigor directly
    field[trtBonus].value += 2 + #trait[attrVig]
    ~equipped armor should add to the Toughness, so we add
that from the armor
    ]]></eval>
  </thing>
```

As indicated in the script above, the effects of armor also need to be added to the "Toughness" trait. However, that must be done separately by each piece of armor that the character has equipped. We can accomplish this within the "Armor" component. Open the file "equipment.str" and take a look at the "Defense" component. There is an Eval script (#2) that applies the effects of equipped armor and shields. In Savage Worlds, shields work differently from armor and don't apply direct adjustments when equipped, so this script should be moved to the "Armor" component. Delete the script from the "Defense" component and then add it to the "Armor" component beneath.

Now we need to modify the script to apply the correct adjustment. Instead of adding the defense rating of the armor to the "trDefense" trait, it needs to be added to the "trTough" trait, so make the change and we're done. The finished script should look something like below:

```
<eval index="2" phase="PreTraits" priority="5000">
  <before name="Calc trtFinal"/>
  <after name="Equipped"/><![CDATA[
  ~if this gear is not equipped, skip it
  if (field[grIsEquip].value = 0) then
    done
    endif
   ~apply the appropriate trait adjustments for the equipment
  #traitbonus[trTough] += field[defDefense].value
  ]]></eval>
```

According to the rules, only armor on the torso of the character is applied to the "Toughness" trait. However, we have no way of determining that yet, so we'll have to add that detail later when we work on equipment. Situations like this will come up at times when developing your data files. The easiest way to handle it is to make a note so that we remember to go back and handle it later.

PARRY TRAIT

The "Parry" trait is very similar to the previous trait. Since the "Parry" trait is combat related, we assign it tags to ensure it gets displayed in appropriate places pertaining to combat. Other than that, the key item of interest for the "Parry" trait is its calculation. The rules stipulate that "Parry" is calculated as "2 plus half your Fighting skill", or simply the value 2 if the character has no "Fighting" skill. We're going to defer handling any overage beyond a "d12" rating until later. So the interesting detail here is that the character might not actually possess the "Fighting" skill.

Normally within HL, if a pick is accessed by a script that does not exist on the character, the line of script code is aborted as invalid and a runtime error is reported. We don't want that to occur. One option would be to write the code to check for the existence of the pick and proceed based on its presence. But there will likely be lots of places within your data files where you would need to add code like this, so something more convenient would be much better.

To support this, HL provides the "childfound" context transition in addition to the "child" transition. The "child" transition reports the error described above, but the "childfound" transition quietly returns a value of zero for a pick that isn't found. This means that we can simply use the "childfound" transition and have everything work exactly as we want in both cases.

To make things even easier, the Skeleton files pre-define a script macro of "#traitfound" that we can use to conditionally access the value of a trait, so all we need to do is use the macro. This results in an Eval script for "Parry" that should look something like what is shown below:

```
 <eval index="1" phase="Traits" priority="4000"><![CDATA[
   ~parry is 2 plus half the Fighting skill, but we track all skills at
the half
   ~value (2-6), so we simply add the Fighting skill that is already
halved; we
   ~use "#traitfound" in case the character does not possess the
Fighting skill
   field[trtBonus].value += 2 + #traitfound[skFighting]
   ]]></eval>
```

CHARISMA TRAIT

The "Charisma" trait is similar to the previous ones, but it has its own wrinkle that needs to be handled. In the interest of efficiency, we'll focus on that one wrinkle here. The wrinkle is that the final Charisma trait value needs to be applied as a bonus/penalty to Persuasion and Streetwise rolls. We can't do this until after the derived trait value is calculated, so we need to add a separate Eval script to the "Charisma" trait that performs this task. We can use the "#traitroll" script macro that we defined earlier to access the appropriate adjustment fields. This results in the following script being added to the "Charisma" trait:

```
<eval index="2" phase="Final" priority="1000">
  <before name="Calc trtDisplay"/><![CDATA[
   #traitroll[skPersuade] += field[trtFinal].value
   #traitroll[skStreet] += field[trtFinal].value
   ]]></eval>
```

PACE TRAIT

The final derived trait for Savage Worlds is "Pace", and, as with Charisma, we'll focus only on its differences here. The "Pace" trait is

different from the others in that it can be adjusted due to some effects and set to an explicit value by other effects. For example, the "Dwarven" race assigns a lower starting "Pace" to a character, and hindrances like "Lame" set an even lower starting "Pace". So we can't just assume everything is an adjustment - we need to handle assignment as well.

The simplest way to handle this is to break up the processing into separate stages, with each stage being assigned to an appropriate evaluation timing. The default value of "6" for "Pace" can be assigned as the default field value for the "trtBonus" field, replacing the need for an Eval script to calculate the value. After that, there are three basic stages that need to be supported, each in order. First is the racial selection, so we pick an appropriate timing such as Setup/1000 for when this should occur. Next is effects that set a fixed value and override any racial effects, such as the "Lame" hindrance, so we pick a later timing such as Setup/10000 for these effects. Lastly, there are the adjustment effects, which can be applied at any time after the previous two.

The last thing of interest with the "Pace" trait is that the value can never drop below one. As such, we need an Eval script to enforce this rule. The script must occur immediately after the "Derived" component calculates the new trait value - before any other script accesses the final trait value.

We can now put it all together and get the following for the "Pace" trait:

```
<thing
  id="trPace"    name="Pace"
  compset="Trait"    isunique="yes"
  description="Description goes here">
  <fieldval field="trtAbbrev" value="Pace"/>
  <fieldval field="trtBonus" value="6"/>
  <tag group="explicit" tag="6"/>
  <tag group="DashTacCon" tag="Combat"/>
  <eval index="2" phase="Traits" priority="6001">
    <after name="Derived trtFinal"/><![CDATA[
    if (field[trtFinal].value <= 0) then
      field[trtFinal].value = 1
      endif
    ]]></eval>
  </thing>
```

RESOURCES (SAVAGE)

There are a number of facets to the Savage Worlds game system that entail the tracking of a resource. Since those game mechanics are relatively fundamental and will likely come into play soon, we might as well get those handled now. The Skeleton files provide two re-usable components and component sets that we can build upon, depending on what we need for a given resource - there is the "Resource" and the "Tracker", and both will be put to good use. These components are defined in the files "miscellaneous.str" and "components.core", respectively. By convention, all resource and tracker things are defined in the file "thing_miscellaneous.dat".

NUMBER OF EDGES

We'll start with something easy. Characters begin the game with the ability to choose one or more edges. The number of edges allowed is driven by a number of effects, such as the character's race. In addition, as characters advance, they may purchase access to new edges. So the number will potentially increase over time.

The number of edges is not directly controlled by the user. It is indirectly affected by selections the user makes. As such, edges are best modeled using the "Resource" mechanism. In fact, all that's needed is to setup the edge count as a "Resource" and simply use all of the built-in capabilities of that component set.

Open the file "thing_miscellaneous.dat" and locate the "resCP" resource that is provided. We'll use that as our template and create a new resource with the unique id "resEdge". Since all characters start out with zero edges by default, we don't need an Eval script to setup the starting quantity, so the script can be deleted. However, all characters need to track an edge count, so we need to ensure the resource gets added to every character. This last requirement is achieved by assigning the "Helper.Bootstrap" tag to the thing.

The net result is a resource defined similarly to the following:

```
<thing
  id="resEdge"
  name="Edges"
  compset="Resource">
  <fieldval field="resObject" value="Edge"/>
  <tag group="Helper" tag="Bootstrap"/>
  </thing>
```

ATTRIBUTE POINTS AND SKILL POINTS

Savage Worlds characters have a starting pool of points that can be spent to increase their initial attributes and skills. There are separate pools for attributes and skills. As such, we need to define a new "Resource" for each of these pools. As with edges, the starting quantity is fixed, so no Eval script is needed, and we need to ensure each resource is added to every character. This results in the following two resources being defined:

```
<thing
  id="resAttrib"
  name="Attribute Points"
  compset="Resource">
  <fieldval field="resObject" value="Attribute"/>
  <fieldval field="resMax" value="5"/>
  <tag group="Helper" tag="Bootstrap"/>
  </thing>

<thing
  id="resSkill"
  name="Skill Points"
  compset="Resource">
  <fieldval field="resObject" value="Skill"/>
  <fieldval field="resMax" value="15"/>
  <tag group="Helper" tag="Bootstrap"/>
  </thing>
```

HINDRANCE POINTS

When a character selects a hindrance, he gains one or two points of hindrance that can be spent to gain offsetting benefits. These points then need to be spent. Consequently, these points are an ideal candidate for implementation as a "Resource". Just like the resources above, it's very simple, except for one key difference. Since this resource will be shown to the user as governing the selection of rewards to offset hindrances, we need to assign a suitable name to show to the user (e.g. "Reward"). The net result looks like below:

```
<thing
```

```
id="resHinder"
name="Hindrance Points"
compset="Resource">
<fieldval field="resObject" value="Reward"/>
<tag group="Helper" tag="Bootstrap"/>
</thing>
```

BENNIES

The Savage Worlds game system has the interesting mechanic of "Bennies". The expenditure of Bennies is controlled directly by the user during the game, so we need to use the "Tracker" mechanism. We'll set the maximum value for the tracker to be the number of Bennies that the character starts each game with. However, we also need to do some special handling for Bennies that entails we define a new component and component set.

It would be beneficial to show the Bennies on the "Special" tab so that the player has the info readily available. This entails including the "SpecialTab" component within the component set, defining a new tag to govern the sequencing of the info on the tab, and assigning the tag to the thing. The assignment can be accomplished within the component, thereby keeping the thing itself very simple.

We'll start by defining the new tag that's required, which takes us to the file "tags.1st". The tag must be added to the "SpecialTab" tag group, and we'll call it "Bennies" for clarity. Tags in this group are assigned an explicit sequencing order, so we must be sure to assign the "order" attribute with a suitable value.

Next, we'll define the new "Bennies" component, which we can add to the file "miscellaneous.str".We must make sure the new "SpecialTab" tag is assigned, so we'll do that in the component. In addition, the default logic for showing information on the "Special" tab is to just give the name of the thing. Since it would be much more useful to show the actual number of Bennies per game session, we need an Eval script to synthesize a customized name for display. The net result is a component that looks something like the following:

```
<component
  id="Bennies"
  name="Bennies"
  autocompset="no">
  <tag group="SpecialTab" tag="Bennies"/>
  <eval index="1" phase="Render" priority="5000"><![CDATA[
    field[spcName].text = "Bennies (" & #trkmax[trkBennies] & "
per game)"
    ]]></eval>
  </component>
```

Once the component is defined, we can define the new component set that builds on top of it (again, within "miscellaneous.str"). Our component set needs to include the standard mechanisms of the "Tracker" component and the new "Bennies" component. We also need to include the "SpecialTab" component so that it will be properly handled for display on the "Special" tab. This translates to the following component set definition:

```
<compset
  id="Bennies">
  <compref component="Tracker"/>
  <compref component="SpecialTab"/>
  <compref component="Bennies"/>
  </compset>
```

The last step involved is adding the actual "Bennies" thing to the data files. This can be added to "thing_miscellaneous.dat", just like the other resources above. Since most of the logic is handled by the underlying components, the "Bennies" thing is relatively simple. We need to set appropriate bounds for the value, and we need to make sure that it's added to every character. We also need to designate the tracker as resetting its value to be the maximum instead of the minimum (which is the default behavior). Lastly, Bennies are accrued in open-ended fashion, so there cannot be a hard limit to the number possessed at any time. Consequently, we need to stipulate that Bennies have no maximum bounding behavior. The result is shown below.

```
<thing
  id="trkBennies"
  name="Bennies"
  compset="Bennies">
  <fieldval field="trkMin" value="0"/>
  <fieldval field="trkMax" value="3"/>
  <tag group="Helper" tag="Bootstrap"/>
  <tag group="Helper" tag="ResetMax"/>
  <tag group="Helper" tag="NoMaxBound"/>
  </thing>
```

NOTE! Since there is only one "Bennies" thing, we could have just as easily defined some of the logic for Bennies on the thing instead of the component. For example, the script and tag could instead be defined on the thing. There is no "right" way in a situation like this. However, as a general rule, we recommend putting the logic on the component whenever practical. There will be times when you decide to add a second thing of a particular type, at which point having the logic on the component will come in handy by eliminating the need to replicate the logic on the new thing.

STARTING CASH

While not maintained by an actual resource object, the cash possessed by characters is an important resource that needs to be tracked. The Skeleton files provide all the mechanics for tracking cash, so we don't have to do much more than configure the built-in mechanisms properly. A character's starting cash is tracked via the "acCashCfg" field within the "Actor" component in the file "actor.str". Characters in Savage Worlds start with a default of $500 in starting cash to spend. We simply need to change the default value for this field to "500" so that characters start with the proper amount of cash.

CORE GAME SYSTEM ELEMENTS

BASICS OF ARCANE BACKGROUNDS (SAVAGE)

Arcane backgrounds are a rather fundamental aspect of the Savage Worlds game system. As such, we should setup handling of the basics for arcane backgrounds early on in our development. We don't need to fully implement them until later, but we do need to handle the basics, as there are a variety of edges and potentially other game elements that influence arcane backgrounds for a character.

ARCANE BACKGROUND TYPES

The first thing we need to track is the different types of arcane backgrounds. The core book defines an initial set of five, and supplements define a number more. Since there will be lots of dependencies on the various arcane disciplines, we need maximum flexibility and the ability to associate the power type with various

kinds of things. So we need to define a new tag group for this purpose, with separate tags for each arcane background. We should also make the tag group dynamic so that the material from supplements can extend this list of arcane backgrounds externally. The net result is a tag group that we can add to the "tags.1st" file and that looks like the following:

```
 <group
   id="Arcane"
   dynamic="yes">
   <value id="Magic"/>      <!-- arcane background is Magic -->
   <value id="Miracles"/>   <!-- arcane background is Miracles --
>
   <value id="Psionics"/>   <!-- arcane background is Psionics --
>
   <value id="SuperPower"/> <!-- arcane background is Super
Powers -->
   <value id="WeirdSci"/>   <!-- arcane background is Weird
Science -->
   </group>
```

Resources to Track

Arcane backgrounds introduce two new resources that need to be managed for each character - the number of arcane powers granted and the pool of power points that can be used to activate the powers. The number of arcane powers available is dictated by the arcane background and edges selected, with the resource decreased as powers are selected by the user. Consequently, the number of powers is best modeled using a "Resource", which we'll call "resPowers". Since every character can have an arcane background, the resource should always be bootstrapped for each character. This resource is very similar to the ones we've already shown in the examples above, so we won't duplicate it here. You can take a look at the completed Savage Worlds data files if you wish to see the specifics.

In contrast, the number of power points will be managed by the user. The total number of power points available will be dictated by the arcane background and edges selected, but the expenditure and recovery of power points will occur during game play. As such, power points are best modeled using a "Tracker". Like the number of powers, every character should always have this tracker bootstrapped. The Skeleton files include a "trkPower" tracker that is perfect for our needs, so we can simply re-purpose it. Since each character starts with zero power points and other sources will influence that number, we can eliminate the script and let the tracker default to values of zero.

Arcane Backgrounds and Skills

The final thing we need to deal with to get the basics of arcane backgrounds setup is adding the various backgrounds and arcane skills. Edges confer access to the different backgrounds, and certain edges are dependent upon the arcane skills, so we need to get arcane backgrounds and skills into place first. In order to keep everything conveniently located in a centralized place, we opt to create a new "thing_arcane.dat" file for all things related to arcane backgrounds.

Before we can do that, though, we first need to define a suitable component and component set to represent arcane backgrounds. We don't need to worry much about how arcane backgrounds work for now, so we'll setup the obvious mechanics and worry about the rest later on. There are three facets of each arcane background that are fundamental: the arcane skill, the starting number of powers, and the starting number of power points. The proper skill will be controlled via each arcane background, so we don't need to deal with that in the component, and the two starting values can be defined as static fields. Since arcane backgrounds don't really fit anywhere else, we'll put the new component and component set in "miscellaneous.str". The two new elements can be handled quite simply for now, and they should look similar to the following:

```
<component
  id="Arcane"
  name="Arcane Background"
  autocompset="no">
  <field     id="arcPowers"
   name="Number of Powers"
   type="static">
   </field>
  <field
   id="arcPoints"
   name="Number of Power Points"
   type="static">
   </field>
  </component>

<compset
  id="Arcane">
  <compref component="Arcane"/>
  </compset>
```

Now that the component set is in place, we can define the individual arcane backgrounds and skills. As outlined above, each background needs to specify the proper fields and assign a suitable tag for the background. In addition, the corresponding arcane skill must be defined for each background. Each skill works just like the standard skills we added earlier. The net result is something that looks like below for the "Magic" arcane background and skill.

```
<thing
  id="arcMagic"
  name="Magic"
  compset="Arcane"
  isunique="yes"
  description="Description goes here">
  <fieldval field="arcPowers" value="3"/>
  <fieldval field="arcPoints" value="10"/>
  <tag group="Arcane" tag="Magic"/>
  </thing>

<thing
  id="skSpellcst"
  name="Spellcasting"
  compset="Skill"
  isunique="yes"
  description="Description goes here">
  <fieldval field="trtAbbrev" value="Spl"/>
  <tag group="Arcane" tag="Magic"/>
  <link linkage="attribute" thing="attrSma"/>
  </thing>
```

Each arcane background defines the appropriate number of starting powers and power points, but we aren't doing anything with those values yet. We need to add the values to the resource and tracker that we just defined above, as they actually manage powers and power points, respectively. This can be accomplished by defining an Eval script on the component for the purpose. All the script needs to do is add the two values to the proper places, which yields something that looks like the example below.

```
<eval index="1" phase="Setup" priority="5000"><![CDATA[
```

```
#resmax[resPowers] += field[arcPowers].value
#trkmax[trkPower] += field[arcPoints].value
]]></eval>
```

There is one last detail that we need to address. When a character has an arcane background, it needs to be known at the top-level in a way that everything within the data files can check on it and base behaviors on it. This means having a tag on the actor that corresponds to the arcane background. Since each background will have a suitable "Arcane.?" tag, the component can forward the tag to the actor. This can be accomplished by adding the Eval script below to the "Arcane" component.

```
<eval index="2" phase="Setup" priority="5000"><![CDATA[
  perform forward[Arcane.?]
  ]]></eval>
```

Races and Racial Abilities (Savage)

The next thing we should probably address is support for races and racial abilities. The original core rulebook for Savage Worlds included a variety of different races, but the Explorer's Edition simply offers the "Human" race. Since races will be a factor in various settings, we'll address them here and focus on the races from the original core rulebook.

Component and Component Set

The Skeleton files already have a suitable component and component set in place for managing races ("Race"), as well as an "Ability" component and component set that can be used for racial abilities. The only change we should consider is that the "Ability" component set includes the "Trait" component, and racial abilities in Savage Worlds don't have any need for those mechanics. Consequently, we can readily delete the "Trait" component from the "Ability" component set in the file "traits.str", eliminating the unnecessary overhead.

```
<compset    id="Ability">
  <compref component="Ability"/>
  <compref component="SpecialTab"/>
  </compset>
```

Adding a Race

Since everything is now in place, we can begin adding the various races and their abilities. In the interest of keeping everything conveniently grouped in a single place, we'll define both races and their abilities in the "thing_races.dat" file. Each race will simply bootstrap all of the abilities that it confers. Those abilities will then possess scripts, as necessary, to apply the appropriate changes to the character.

We'll start with a race that is relatively simple, the "Atlanteans". This race has two abilities, with one ability conferring a bonus that is applied via an Eval script. Since the "Aquatic" ability is specially tailored for the race, we have to anticipate the need for a different "Aquatic" ability for a different race, so the unique id should be tied to the race. A generalized ability like "Tough" should have a generic id that can be readily re-used by another race. The three pieces should look something like below:

```
<thing
  name="Atlantean"
  compset="Race"
  isunique="yes"
  description="Description goes here">
  <bootstrap thing="abAquaAtl"/>
  <bootstrap thing="abTough"/>
  </thing>

<thing
  id="abAquaAtl"
  name="Aquatic"
  compset="Ability"
  isunique="yes"
  description="Description goes here">
  </thing>

<thing
  id="abTough"
  name="Tough"
  compset="Ability"
  isunique="yes"
  description="Description goes here">
  <eval index="1" phase="PreTraits" priority="5000">
    <before name="Calc trtFinal"/><![CDATA[
    ~add +1 die to the Vigor attribute
    #traitbonus[attrVig] += 1
    ]]></eval>
  </thing>
```

Special Case Situations

Among all of the races outlined in the Savage Worlds rulebook, there are a handful of special case situations that warrant specific commentary. Each of these is outlined in this section. Except for these special cases, the races are rather straightforward.

For the "Dwarves" race, there is the "Slow" ability, which sets the base "Pace" value for the Dwarf to five instead of the normal six. In order for this to work properly, be sure to set the appropriate value for "Pace" using the conventions established earlier in this walk-through. An example of the "Slow" ability is presented below.

```
<thing
  id="abSlowDwf"    name="Slow"
  compset="Ability"
  isunique="yes"
  description="Description goes here">
  <!-- Dwarves have a base Pace of 5
          NOTE! Since we're setting the Pace to a fixed value,
do it early, before
                  any adjustments are applied.
  -->
  <eval index="1" phase="Setup" priority="1000"><![CDATA[
    #traitbonus[trPace] = 5
    ]]></eval>
  </thing>
```

The "Elves" race possesses the "All Thumbs" ability, which is actually the "All Thumbs" hindrance. There are two options for how we can handle this properly. The first option is to duplicate the behaviors of the "All Thumbs" hindrance as a separate ability, and the second is to automatically bootstrap the "All Thumbs" hindrance from the ability. The latter approach requires that the bootstrapped hindrance be specially implemented so that it does not confer bonus points that can be traded in for edges or other benefits. That extra handling is more work, so it's simpler to just duplicate the "All Thumbs" hindrance as a racial ability.

Unfortunately, having a separate "All Thumbs" ability introduces another wrinkle that needs to be resolved. Since the "All Thumbs" ability is equivalent to the "All Thumbs" hindrance, it's important that the user not be allowed to take the hindrance when the "Elves" race is selected for a character, and vice versa. Actually precluding this is a bit of work, but it's relatively easy to detect when the user has done it and report a validation error, so that's the method we'll employ. Sadly, we can't implement our solution yet, since we don't have the hindrance in place yet, so we'll make a note to come back and deal with this later.

Next on the list of special cases is the "Heritage" ability of Half-Elves. This ability gives the character the ability to choose one of two different options: the "Agile" ability of Elves or a free edge of the player's choice. A variety of different methods could be employed to solve this, ranging from simple to complex. We're going to keep things simple, so we'll create two separate versions of the race, with each conferring a different option. This way, the user simply selects the version he wants and everything works easily. The names of each race must include an indication of the benefit being granted, such as "Half-Elven (Edge)", which is shown below.

```
<thing
  id="racHfElf1"
  name="Half-Elven (Edge)"
  compset="Race"
  isunique="yes"
  description="Description goes here">
  <bootstrap thing="abHeritHE"/>
  <bootstrap thing="abFreeEdge"/>
  <bootstrap thing="abLowLight"/>
  <bootstrap thing="abOutHE"/>
  </thing>
```

Although a few different races have the "Outsider" ability, each instance of the ability is a bit different from the others. As such, we can't just have a single "Outsider" ability that can be re-used. We must instead have multiple abilities that all have the same name yet differ in their details.

The final special case is that a couple of races grant a natural weapon of some sort. The Rakashans have a "Claws" ability that behaves as a natural weapon, while the Saurians have an explicit "Natural Weapon" ability. We aren't able to add weapons yet, so we need to make a note about these and come back to implement them later.

HUMANS AS DEFAULT

The Explorer's Handbook centers solely on humans. Consequently, we have to assume most players will be starting with human characters. We could make the player always select a race, or we could select "Human" as the default race and let the player change it if he wants. We'll opt for the latter approach.

Defaults choices are generally quite easy to handle with the Kit. You can automatically add a thing to table or set up a chooser with a default selection. This is achieved via an "autoadd" element and is performed within the file "bootstrap.1st". The element specifies both a thing to be added and the portal it must be added to. The Kit handles everything else. Auto-selecting the "Human" race results in the following XML element being added to the file.

```
<autoadd thing="racHuman" portal="stRace"/>
```

That's all there is to it. Whenever a new character is created, it will begin with the "Human" race as the default selection.

VERIFYING ACTOR PRE-REQUISITES (SAVAGE)

Before we can successfully implement hindrances and edges, there are a few facets of the actor that must be tracked. These facets behave as pre-requisites and are tested by certain hindrances and edges, so they need to be managed properly.

CHARACTER RANK

Every edge specifies the minimum rank required for a character to select it, and the rank is based on the number of experience points earned by the character during play. This rank needs to be calculated and tracked for the actor so that the edges can verify compliance properly. In order to do this, we need to add a new field to the "Actor" component that represents the rank and auto-calculates itself. We use a numeric value to represent the rank, since numbers are easily compared, so the five character ranks defined in the rulebook are assigned a value from 0-4, with zero representing a Novice (starting) character and four indicating a Legendary character.

Open the file "actor.str" and the first component is the "Actor" component. Towards the end of the list of fields, add a new field for the rank. Define a suitable Calculate script for the field and it should be all set. The resulting new field should look similar to the following:

```
<field
  id="acRank"
  name="Current Rank"
  type="derived">
  <calculate phase="Final" priority="1000"><![CDATA[
    var xp as number
    xp = hero.child[resXP].field[resMax].value
    if (xp < 80) then
      @value = round(xp / 20,0,-1)
    else
      @value = 4
      endif
    ]]></calculate>
  </field>
```

WILD CARDS

Quite a few edges are only applicable to characters that are designated as "wild cards". As such, this state also needs to be tracked within the "Actor" component. We need to allow the user to control this state, so it needs to be a "user" field. To keep things simple, we consider a value of zero to indicate the character is not a wild card, while a non-zero value indicates the character is a wild card. Since the majority of characters that users will create will be wild cards, we assume a value of one as our default. In the end, the field should look a lot like the following:

```
<field
  id="acIsWild"
  name="Is Wild Card?"
  type="user"
  defvalue="1">
  </field>
```

ENCUMBRANCE (SAVAGE)

In reviewing all of the hindrances and edges, there is an additional game mechanic that edges can influence: encumbrance. Consequently, this mechanic should be implemented before we start work on hindrances and edges.

LOAD LIMIT MULTIPLIER

A key facet of encumbrance in Savage Worlds is the "load limit", which represents the maximum load a character can carry before incurring penalties. It is derived from the "Strength" attribute and a multiplier. By default, characters have a multiplier of five, but edges can modify this multiplier. Consequently, the multiplier needs to be tracked on the "Actor" component as a field that can be changed dynamically. It should look something like the following:

```
<field
  id="acLoadMult"
  name="Load Limit Multiplier"
  type="derived"
  defvalue="5">
  </field>
```

NEW RESOURCES

Once the multiplier is in place, we then need to track the actual limit itself. However, the limit depends on the actual encumbrance accrued to the character. So we really need to track both the encumbrance and the load limit as separate items. These are best modeled via "Resource" components, with both calculating their effective maximum limit. Gear carried by the character is accrued into the encumbrance resource as an amount spent. This amount is then shared into the load limit resource, making it easy to automatically determine whether the character has exceeded the load limit, and by how much. If the limit is exceeded, then the appropriate penalties can be applied to the various trait rolls via a script. Similarly, the calculated load limit is shared into the encumbrance resource to determine the next encumbrance threshold for display to the user. This yields a pair of resources that look like the following:

```
<thing
  id="resEncumb"
  name="Encumbrance"
  compset="Resource">
  <fieldval field="resMin" value="0"/>
  <tag group="Helper" tag="Bootstrap"/>

 <eval index="1" phase="Final" priority="500">
   <before name="Calc resLeft"/><![CDATA[
   ~get our limit from the load limit resource
   var limit as number
   limit = #resmax[resLoadLim]

   ~calculate how many multiples we've consumed via the load
limit details
   var multiples as number
   multiples = 1
   if (field[resSpent].value >= limit) then
     multiples += round(field[resSpent].value / limit,0,-1)
     endif

   ~set our maximum appropriately for encumbrance reporting
   field[resMax].value = multiples * limit
   ]]></eval>
 </thing>

<thing   id="resLoadLim"
   compset="Resource">
   <fieldval field="resMin" value="0"/>
   <tag group="Helper" tag="Bootstrap"/>
  <eval index="1" phase="Traits" priority="7000" name="Calc
LoadLimit">
    <before name="Calc resLeft"/><![CDATA[
    ~our maximum is based on the Strength and load limit
multiplier
    field[resMax].value = herofield[acLoadMult].value *
#trait[attrStr] * 2
    ]]></eval>

  <eval index="2" phase="Traits" priority="8000" name="Apply
LoadLimit">
    <after name="Accrue Weight"/>
    <after name="Calc LoadLimit"/><![CDATA[
    ~get quantity spent from encumbrance resource and save it
in "extra" field
    var spent as number
    spent = #resspent[resEncumb]
    field[resExtra].value = spent

   ~if we haven't exceeded the load limit, there's nothing to do
    if (spent <= field[resMax].value) then
      done
      endif

   ~calculate how many multiples we exceeded the limit by
    var overage as number
    overage = round(spent / field[resMax].value,0,-1)

   ~apply the appropriate penalty to all Strength and Agility rolls
    #traitroll[attrStr] -= overage      #traitroll[attrAgi] -= overage

   ~apply the penalty to all skills based on Strength and Agility
    ~???? we can't do this yet ????
    ]]></eval>
  </thing>
```

Calculating and applying the penalties for exceeding the limit can all be handled via a single script. The interesting detail is that we need to apply the penalty to all skills that are linked to the "Strength" and "Agility" traits. Doing this will require that we make a few enhancements to the data files, which are covered in the next section.

IDENTIFYING SKILLS TIED TO ATTRIBUTES

We need to identify which skills are tied to which attributes. The easiest way to do this is to assign a tag to each skill that identifies the attribute it's associated with. So we need to define a new tag for each attribute that uniquely identifies that attribute. We can do this ourselves, or we can let HL do all of the work for us. We'll opt for the latter approach and modify the "Attribute" component to have an "identity" tag group named "Attribute". This will automatically define an "identity" tag for each attribute in the data files. Each tag will have the unique id of the thing, so the "Strength" attribute will have an identity tag of "Attribute.attrStr". We can accomplish all this by adding the single line shown below to the "Attribute" component:

```
<identity group="Attribute"/>
```

We now have an identity tag defined for every attribute, with that tag being automatically assigned to each attribute. However, we still need to get the tag onto the skill. We could do this manually by adding the tag to each skill within its definition, but that is tedious and error prone. Each skill already has a linkage to its associated attribute, and we can access the linked attribute via a script, so we

can easily write a script for skills that pulls the identity tag out of the attribute and assigns it to the skill. Since the behavior is the same for every skill, we can add this logic to the "Skill" component and let HL do all of the work for us. The new script for the "Skill" component should look something like the following:

```
<eval index="2" phase="Setup" priority="5000"><![CDATA[
  ~pull the identity tag of the linked attribute into the skill
  perform linkage[attribute].pullidentity[Attribute]
  ]]></eval>
```

Revising the Resource

At this point, every skill should be automatically assigned the appropriate "Attribute" tag for the linked attribute. Now we can leverage this information to get our resource working completely. We need to identify all skills that are tied to the "Strength" or "Agility" attributes and operate on each of them. This is achieved by using the Kit's "foreach" statement within scripts, which iterates through all picks that match a given tag expression and allows you to process them.

We need to determine the tag expression to use. First of all, we need to limit ourselves to processing skills. Every thing in the game system is automatically assigned one or more tags from the "component" tag group. Each component has a tag defined and each thing is assigned a tag for each component it derives from. Therefore, every skill will possess the "component.Skill" tag. In order to limit the proper attribute tags, we can check for each of those explicitly. Putting it all together, we end up with a tag expression that identifies a skill that is also either tied to "Strength" or "Agility", as shown below:

```
component.Skill & (Attribute.attrStr | Attribute.attrAgi)
```

With the tag expression figured out, we can now write the script code to locate each skill we want and apply the appropriate penalty. We process the picks within the "hero", since all skills are assigned directly to the hero (i.e. character), and we apply the penalty to each one we access using the "eachpick." transition. The resulting script code is shown below and can be spliced into the resource where we left the big question mark a moment ago.

```
foreach pick in hero where "component.Skill & (Attribute.attrStr |
Attribute.attrAgi)"
  eachpick.field[trtRoll].value -= overage
  nexteach
```

Hindrance Support (Savage)

Now that we've got all the pieces in place that are depended upon, we can add support for hindrances, which are disadvantages that can be optionally selected for a character.

Component and Component Set

While hindrances have a lot of behaviors in common with the existing "Ability" component set, they also introduce a variety of new behaviors that must be properly managed. As such, we need to define a new "Hindrance" component and a corresponding component set. We'll flesh out the new component in the following sections, so we'll start by defining a simple component that does nothing as an initial framework. Then we can define our component set appropriately, building upon the "Ability" and "SpecialTab" components as well. The net result is the following:

```
<component
  id="Hindrance"
  name="Hindrance"
  autocompset="no"
  panellink="abilities">
  </component>

<compset   id="Hindrance">
  <compref component="Hindrance"/>
  <compref component="Ability"/>
  <compref component="SpecialTab"/>
  </compset>
```

Inclusion on "Special" Tab

Since hindrances will come up during play, we want to make sure they appear on the "Special" tab. The basic appearance is handled by inclusion of the "SpecialTab" component in the component set, but we need to customize the behavior appropriately. First of all, we need to define a new classification tag for ordering the display of hindrances on the "Special" tab. This is solved by adding a "Hindrance" tag to the "SpecialTab" tag group within the "tags.1st" file, being sure to specify a suitable order value to sequence everything appropriately.

```
<value id="Hindrance" order="30"/>
```

Once the tag is defined, we then need to make sure that all hindrances are assigned the tag. We can do that within the component by simply adding the line below to the component:

```
<tag group="SpecialTab" tag="Hindrance"/>
```

Major Versus Minor

Hindrances can be classified as either major or minor in nature. Some hindrances are always major, others are always minor, and there are some where the user can decide whether the severity is major or minor. So we need to define a field for hindrances to track the severity of the hindrance. Since the severity is a simple two-state condition, we use a value of zero to indicate a minor hindrance and a value of one to indicate something major. The result is shown below.

```
<field
  id="hinMajor"
  name="Is Major?"
  type="user"
  minvalue="0"
  maxvalue="1">
  </field>
```

We also need to handle the distinction between a hindrance with a fixed severity and one that is user-selectable. We can use either a field or tag for this, but a tag is easier because we can always detect a tag within a tag expression, so we'll use a tag this time. We only need one tag, since the tag indicates one behavior and the lack of the tag indicates the other. So we define a new tag in the "User" tag group to indicate a hindrance is user-selectable and give it the "UserSelect" unique id. Any hindrance that is user-selectable must be assigned this tag as part of its definition.

```
<value id="UserSelect"/>  <!-- Whether a hindrance is user-selectable for major/minor (vs. fixed)  -->
```

User-Specified Domain

Certain hindrances are similar to the "Knowledge" skill in that they require the user to specify a "domain" in which the hindrance applies. For example, the "Phobia" and "Habit" hindrances each need to identify the subject of the phobia or the type of habit. Just like we did for skills, we need to integrate the "Domain" component into the component set so that the user-specified domain can be tracked for appropriate hindrances. This is achieved by adding a new "compref" element to the component set that includes the "Domain" component.

We also need to identify which hindrances require the user to provide a domain. Fortunately, we can easily re-use the same "User.NeedDomain" tag that we leveraged for skills. If a hindrance has the "User.NeedDomain" tag assigned, then the user is expected to specify a domain.

If a hindrance has a domain specified, then it would be valuable for that domain to be shown as part of the hindrance on the "Special" tab. Consequently, we should generate a suitable, customized name for displaying the hindrance on the "Special" tab. By default, the "Domain" component already integrates the domain into the name, so we can simply use that default behavior. If we wanted to do something different, we could define an Eval script that sets the "spcName" field to whatever is appropriate.

Hindrance Points

Whenever a hindrance is selected for a character, hindrance points are accrued that can be traded in for offsetting benefits to the character. Each hindrance must therefore add the appropriate number of hindrance points to the resource we created earlier. Since the rule for hindrance points is one point for a minor hindrance and two points for a major hindrance, we can perform this automatically for each hindrance via a script on the component. The script would look something like the one shown below.

```
<eval index="2" phase="Setup" priority="5000"><![CDATA[
  #resmax[resHinder] += field[hinMajor].value + 1
  ]]></eval>
```

Identification for Pre-Requisites

There are a few situations where a particular hindrance conflicts with something else. For example, it's not possible to combine the "Luck" edge with the "Bad Luck" hindrance. We also have to identify when the user tries to take the "All Thumbs" hindrance with the "Elves" race. In order to do this, we need to track which hindrances have been added to the character in a centralized way. This is most easily accomplished by designating an "identity" tag group for hindrances and then forwarding the identity tag of each selected hindrance to the hero. The net result is that the hero will always possess tags for each hindrance chosen for the character.

We saw earlier that designating an identity tag group on a component requires a single line to be added to the component. This automatically defines the tag group and assigns the proper tag to each thing. Then a one-line script can be defined for the component that forwards the identity tag up to the hero. The two pieces should look like the following for hindrances:

```
<identity group="Hindrance"/>

<eval index="1" phase="Setup" priority="5000"><![CDATA[
  perform forward[Hindrance.?]
  ]]></eval>
```

Adding Hindrances (Savage)

Now that all of the mechanics are in place for hindrances, it's possible to actually define them. To keep them all together as a group, we create a new "thing_hindrances.dat" file where they will be added. When adding hindrances, be mindful of using the correct "#traitroll" or "#traitbonus" macro, since the two have important differences in their behavior.

General Reminders

Some hindrances apply penalties that are situational. When the situation may persist for an extended period, the user may want to toggle the effects on and leave them that way for a period of time. To support this, appropriate hindrances should be designated as able to be user-activated by assigning the "User.Activation" tag. This will display the hindrance as something the user can toggle on and off within the "In-Play" tab. A script that applies the effects can then check the actual activation state of the hindrance and apply the effects when appropriate.

If you need to reference a trait from a script of some sort (e.g. within a pre-requisite test), be careful to handle it properly. If the trait is a skill or something else the user can optionally add to the character, the trait might not exist on the character. If that's possible, be sure to use the "#traitfound" macro instead of "#trait", else you will receive run-time errors from HL that you're trying to access picks that don't exist.

Adding all the hindrances is a straightforward process of going through and entering all of the details accurately. We'll select a few representative samples to cover in the sections below, pointing out key facets of each.

Something Simple

We'll start with the "Delusional" hindrance, since it's relatively basic in nature. This hindrance can be either minor or major in severity, so we assign it an initial severity of minor (via the field) and then assign it the "User.UserSelect" tag to indicate the user can select the ultimate severity level. Other than that, there is nothing to do, since the hindrance's effects are something the GM must adjudicate. The result is a simple hindrance as shown below.

```
<thing
  id="hinDelude"
  name="Delusional"
  compset="Hindrance"
  isunique="yes"
  description="Description goes here">
  <fieldval field="hinMajor" value="0"/>
  <tag group="User" tag="UserSelect"/>
  </thing>
```

User Activation

Now we'll look at the "Bloodthirsty" hindrance, since it will demonstrate user-activated behaviors. This hindrance is always major in severity, so we set the field value accordingly. The effects of the hindrance don't always apply, so we allow the user to control

when to apply them by assigning the "User.Activation" tag. In the script that applies the effects, we then check whether the activation state is on or off, applying the effects only if activated. The net result is the definition below.

```
<thing
  id="hinBloodth"
  name="Bloodthirsty"
  compset="Hindrance"
  isunique="yes"
  description="Description goes here">
  <fieldval field="hinMajor" value="1"/>
  <tag group="User" tag="Activation"/>

 <eval index="1" phase="PreTraits" priority="5000">
   <before name="Calc trtFinal"/><![CDATA[
   ~if activated, suffer -4 on Charisma
   if (field[abilActive].value <> 0) then
     #traitbonus[trCharisma] -= 4
     endif
   ]]></eval>
  </thing>
```

User-Specified Domains

The "Phobia" hindrance is a perfect example of when the user needs to specify a suitable domain, in this case identifying what the character is afraid of. As far as the mechanics are concerned here, requiring the user to specify a domain is as simple as assigning the "User.NeedDomain" tag. So that's all we need to do besides allowing the user to designate the severity, and the results can be seen below.

```
<thing
  id="hinPhobia"
  name="Phobia"
  compset="Hindrance"
  isunique="yes"
  description="Description goes here">
  <fieldval field="hinMajor" value="0"/>
  <tag group="User" tag="UserSelect"/>
  <tag group="User" tag="NeedDomain"/>
  </thing>
```

Dynamic Control Over User Activation

Now we'll switch to the "Bad Eyes" hindrance, since it demonstrates how to combine many of the mechanics involved and throws an interesting wrinkle into the mix. This hindrance can be either minor or major, so we assign it an appropriate field value and tag. When the hindrance is minor, the user can toggle the penalties on and off based on whether the character has temporarily lost his glasses, so we should probably assign the "User.Activation" tag to give the user control. But when the hindrance is major, the penalties always apply, in which case we don't want the user to have control. So what we really need to do is have the hindrance be user controlled only when it is minor. This requires that we change the behavior dynamically, which can be done within the script by assigning the tag only when the hindrance is minor. Putting it all together yields the following definition:

```
<thing
  id="hinBadEyes"
  name="Bad Eyes"
  compset="Hindrance"
  isunique="yes"
  description="Description goes here">
   <tag group="User" tag="UserSelect"/>

  <eval index="1" phase="PreTraits" priority="5000">
    <before name="Calc trtFinal"/><![CDATA[
    ~if this is a minor hindrance, allow the user to activate it
    if (field[hinMajor].value = 0) then
      perform assign[User.Activation]
      endif

    ~if this is a major hindrance or the user has activated it, apply
the effects
    if (field[hinMajor].value + field[abilActive].value <> 0) then
      #traitroll[skShooting] -= 2
      #traitroll[skNotice] -= 2
      endif
    ]]></eval>
  </thing>
```

Handling Pre-Requisites

In rare cases, a hindrance will have pre-requisites that need to be properly handled. For example, the "Bad Luck" hindrance can't be taken at the same time as the "Luck" or "Great Luck" edges. For situations like this, we need to define an appropriate pre-requisite test that checks the state of the character and reports whether the pre-requisite conditions are satisfied. If not satisfied during selection display, the invalid choice can be highlighted appropriately. However, if an invalid selection is actually added to the character, then a validation message is displayed to the user. We can also designate the tab panel corresponding to the hindrance as invalid to highlight it to the user. We can put it all together into something that looks like the following:

```
<thing
  id="hinBadLuck"
  name="All Thumbs"
  compset="Hindrance"
  isunique="yes"
  description="Description goes here">
  <fieldval field="hinMajor" value="1"/>
  <eval index="1" phase="PreTraits" priority="5000">
    <before name="Calc trtFinal"/><![CDATA[
    #trkmax[trkBennies] -= 1
    ]]></eval>
 <!—
  <pickreq ispreclude="yes" thing="edgLuck"/>
  <pickreq ispreclude="yes" thing="edgLuck2"/>
 -->
  </thing>
```

NOTE! In the above "pickreq" elements, we check for the existence of one of the two Luck edges. However, these things won't exist until we actually define the edges, so the compiler will reject the above thing as invalid. For now, we'll comment out the two XML elements, as shown above, and make a note to put these back into place when we actually add the edges later.

Acknowledging Our Limitations

Every now and then, you'll come across an incredibly specialized requirement for a particular thing that will require an inordinate amount of work to properly verify within the data files. When this happens, you can either spend many hours to handle the special case situation or you can defer the special case to the user. A perfect example of this within the Savage Worlds data files is the "Elderly" hindrance, which dictates that the extra skill points it confers must be spent on skills based on the "Smarts" attribute. It is simple to increase the number of skill points available to be spent, but

requiring them to all be spent on a subset of skills requires that the additional skills be purchased and tracked separately from the standard pool of skills. While possible, it's a lot of work to implement properly, and all for a lone special case that is probably not very frequently used by players. In situations like this, it's often best to simply acknowledge the special case as a limitation of the data files and flag it to the user within the description text for the thing. That's what we're going to do with the "Elderly" hindrance.

Countering Hindrances (Savage)

When a user selects hindrances for a character, that character gains benefits to offset those hindrances. The hindrance points are accrued, and the user must now redeem those hindrance points.

Component and Component Set

Within the Savage Worlds game system, there are four choices for how to spend the accrued hindrance points, so we need a way to mange this. This entails adding a new component and component set, and we'll refer to each of these options as a "Reward". Each reward is extremely simple, with a field specifying how many hindrance points are redeemed for the reward and all other details being managed through an Eval script.

An appropriate implementation of the component and component set are presented below. Note that the same reward can be selected multiple times, so it's important that rewards not be designated as unique.

```
<component
   id="Reward"
   name="Reward for Hindrances"
   autocompset="no"
   panellink="abilities">
   <field
    id="rewPoints"
    name="Hindrance Points"
    type="static">
    </field>
   <eval index="1" phase="Setup" priority="5000"><![CDATA[
    #resspent[resHinder] += field[rewPoints].value
    ]]></eval>
   </component>

 <compset
   id="Reward"
   forceunique="no">
   <compref component="Reward"/>
   </compset>
```

Adding the Rewards

We'll add all of our rewards into a new file named "thing_rewards.dat", thereby keeping them all together in a convenient location. Three of the rewards entail merely adjusting the maximum value for a resource to confer the benefit. For example, the reward that grants an extra attribute point is shown below. Remember that the same reward can be selected multiple times, so rewards must not be unique.

```
 <thing
   id="rewAttrib"
   name="+1 Attribute Point"
   compset="Reward"
   description="Description goes here">
   <fieldval field="rewPoints" value="2"/>
   <eval index="1" phase="Setup" priority="5000"><![CDATA[
    ]]></eval>
   </thing>
```

The fourth reward doubles the character's starting cash, which is easy in concept but currently cannot be handled by the data files. There is a field on the "Actor" component for tracking the character's starting cash, but this is a user-controlled field. As such, if we double this value in a script, we'll physically change the value that the user can specify. In addition, we'll keep doubling the value again every time that HL needs to re-evaluate everything. So we need to keep the effects of this reward decoupled from the user-controlled starting cash field.

One simple solution would be to revise the Eval script within the "Actor" component that calculates the total cash possessed by the character. We could check for the existence of the "double cash" reward and multiply the starting cash by two if it's found. This could entail defining a tag that the reward could assign to the hero, or the Eval script could explicitly look for the specific reward. While this would work, it would be exclusively tied to this one reward. Given that this is a role-playing game, it's quite likely that some GMs (or potentially some official supplements) will introduce other cases that can influence the character's starting cash. So we should come up with a more generalized solution.

A solid, generalized solution would take the form of adding a new field to the "Actor" component. This new field would track a starting cash multiplier that would default to one. For the reward that doubles the starting cash, the value could be multiplied by two. For another effect that decreases starting cash by 10%, the value could be multiplied by 0.9. The final value of this field could then be integrated into the Eval script that calculates the net cash possessed by the character. Putting it together, the new field and revised Eval script would look like those presented below.

```
  <field
    id="acCashMult"
    name="Starting Cash Multiplier"
    type="derived"
    defvalue="1">
    </field>

  <eval index="3" phase="Effects" priority="5000"><![CDATA[
    ~our net cash is our configured starting cash plus our accrued
cash
    field[acCashNet].value = field[acCashCfg].value *
field[acCashMult].value +
   hero.usagepool[TotalCash].value
    ]]></eval>
```

Once the above mechanics are in place, we can add the fourth reward, and it should look similar to the example shown below.

```
  <thing
    id="rewCash"
    name="Additional Starting Cash"
    compset="Reward"
    description="Description goes here">
    <fieldval field="rewPoints" value="1"/>
    <eval index="1" phase="Setup" priority="5000"><![CDATA[
     herofield[acCashMult].value += 1
     ]]></eval>
    </thing>
```

EDGE SUPPORT (SAVAGE)

In Savage Worlds, edges represent the beneficial counterparts of hindrances, and we now need to get them working.

COMPONENT AND COMPONENT SET

As with hindrances, edges have a lot of behaviors in common with the existing "Ability" component set, but they also introduce new behaviors that we must manage. To that end, we'll define an "Edge" component and a corresponding component set. The new component will be evolved in the following sections, so we start here with a simple component that does nothing and add the component set on top of it, incorporating the "Ability" and "SpecialTab" components as well. Since edges can be selected through advancements, we'll also include the "CanAdvance" component that we'll need later. The net result is the following, which we'll add to the file "traits.str":

```
<component
  id="Edge"
  name="Edge"
  autocompset="no"
  panellink="abilities">

<compset
  id="Edge">
  <compref component="Edge"/>
  <compref component="Ability"/>
  <compref component="SpecialTab"/>
  <compref component="CanAdvance"/>
  </compset>
```

INCLUSION ON "SPECIAL" TAB

The influence of edges will be important during play, so we need to make sure they appear on the "Special" tab. The basic appearance is handled by inclusion of the "SpecialTab" component in the component set, but we need to customize the behavior appropriately. We first need to define a new classification tag for ordering the display of edges on the "Special" tab. To do this, we add an "Edge" tag to the "SpecialTab" tag group within the file "tags.1st" with a suitable order value to sequence everything appropriately.

```
<value id="Edge" order="40"/>
```

Once the tag is defined, we need to make sure that all edges are assigned the tag. We do that within the component by adding the line below:

```
<tag group="SpecialTab" tag="Edge"/>
```

EDGE CATEGORIES

The Savage Worlds rulebook breaks edges up into an assortment of categories. In keeping with that approach, the data files should do the same. The solution is to create a new tag group for the edge category and populate it with tags corresponding to each category. So we create an "EdgeType" tag group and define the appropriate tags, as shown below. Every edge should be assigned one of these tags.

```
<group
  id="EdgeType">
  <value id="Combat"/>
  <value id="Leadership"/>
  <value id="Power"/>
  <value id="Profession" name="Professional"/>
  <value id="Social"/>
  <value id="Weird"/>
  <value id="Wildcard" name="Wild Card"/>
  <value id="Legendary"/>
  </group>
```

IDENTIFICATION FOR PRE-REQUISITES

There are numerous situations where one edge precludes or depends upon another edge. The most common example is the various "levels" of an edge, where there is a basic version of an edge and one or two advanced versions that require the lesser versions to already be taken. For example, the "Very Attractive" edge requires the "Attractive" edge as a pre-requisite. To support this, we must track the edges that have been added to the character in a centralized way. This is most easily accomplished by designating an "identity" tag group for edges and then forwarding the identity tag of each selected edge to the hero. The net result is that the hero will always possess tags for each edge chosen for the character, which we can then test against.

We can designate an identity tag group on a component by adding a single line, which automatically defines and assigns the tag to each thing. Forwarding the identity tag to the hero is achieved with a one-line script. Both pieces should look like the following:

```
<identity group="Edge"/>

<eval index="1" phase="Setup" priority="5000"><![CDATA[
  perform forward[Edge.?]
  ]]></eval>
```

MINIMUM RANK

There is one last facet of edges that we haven't addressed yet. Every edge has a minimum character rank specified, and the character must be at least that rank in order to select the edge. So we need to track the minimum rank, and the easiest way to do it is to treat the rank as a numeric value and use a field to specify the value. As we established previously, character ranks range from 0-4, with zero corresponding to a Novice and four indicating a Legendary character. So we need to add a field for the minimum rank, as shown below.

```
<field
  id="edgMinRank"
  name="Minimum Rank"
  type="static">
  </field>
```

Once we have the field in place, we then need to verify that the character satisfies the minimum rank as a pre-requisite. This is accomplished by defining a shared pre-requisite on the component that will be inherited by all things derived from the component. Within the Valid script for the pre-requisite, we do not need to differentiate between whether it is testing a thing prior to selection or a pick that has been added by the user. The field that dictates the requirement is "static", so it cannot be changed for a pick. Consequently, we can always utilize the state of the thing instead of having to distinguish between picks and things. We also need to

synthesize an appropriate validation error message to report to the user. Putting it all together yields the following pre-requisite:

```
<prereq iserror="yes" message="Minimum Rank required.">
  <valid><![CDATA[
   ~get the minimum rank required for the edge to be valid
   var rank as number
   rank = altthing.field[edgMinRank].value

   ~if the minimum rank is satisfied, we're good to go
   if (herofield[acRank].value >= rank) then
     @valid = 1
     done
     endif

   ~mark the panel as invalid
    altthing.linkvalid = 0

   ~synthesize an appropriate validation error message
   if (rank = 1) then
     @message = "Seasoned"
   elseif (rank = 2) then
     @message = "Veteran"
   elseif (rank = 3) then
     @message = "Heroic"
   elseif (rank = 4) then
     @message = "Legendary"
     endif
   @message &= " rank required."
   ]]></valid>
 </prereq>
```

Wild Card Restrictions

A fair number of edges are limited for use only with "wild card" characters, such as PCs and pivotal NPCs in the story. We could implement a separate pre-requisite test for each of these edges, but the number is significant and it is much easier to handle everything in a very generic way within the component. We start by defining a new field for the component that indicates whether the character is required to be a "wild card", as shown below. A non-zero value indicates that a wild card is required and the default value is zero to indicate no requirement unless explicitly specified otherwise.

```
<field
  id="edgIsWild"
  name="Is Wild Card Needed?"
  type="static">
  </field>
```

We can now define a pre-requisite on the component that compares the requirement field against the field we added earlier on the character that indicates whether the character actually is a "wild card". We do not need to differentiate between picks and things within the Valid script, since the wild card requirement field is static and cannot change for a pick. The net result is shown below.

```
<prereq iserror="yes" message="Wild Card required.">
  <valid><![CDATA[
   ~if the hero is a wild card, then we know we're going to be
valid
   if (herofield[acIsWild].value <> 0) then
     @valid = 1
     done
     endif

   ~we're valid if the edge does NOT require a wild card
   if (altthing.field[edgIsWild].value = 0) then
     @valid = 1
     endif

   ~we're not valid, so mark the panel as invalid
    altthing.linkvalid = 0
   ]]></valid>
 </prereq>
```

Edge Slots

Every character has a dynamic number of edges that he is allowed to select, as dictated by the "resEdge" resource that we defined earlier. Whenever an edge is selected for a character, one slot must be spent from the resource. We can perform this automatically for each edge via a script on the component, and the script should look something like the one shown below.

```
<eval index="2" phase="Setup" priority="5000"><![CDATA[
  #resspent[resEdge] += 1
  ]]></eval>
```

Adding Edges (Savage)

With the mechanics for edges in place, we now need to get them implemented. We'll keep them all together as a group by creating a new "thing_edges.dat" file where they will be added. All of the reminders outlined for hindrances also apply to edges, and the following sections discuss a few key edges to highlight important facets of their implementation.

Something Simple

We'll start with the simple "Combat Reflexes" edge. This edge is of type "Combat" and has a minimum rank of "Seasoned" (or one), so we assign the tag and set the field value accordingly. The result is shown below.

```
<thing
  id="edgComRefs"
  name="Combat Reflexes"
  compset="Edge"
  isunique="yes"
  description="Description goes here">
  <fieldval field="edgMinRank" value="1"/>
  <tag group="EdgeType" tag="Combat"/>
  </thing>
```

User Activation

Next up is the "Berserk" edge, which demonstrates the handling of user-activated behaviors. The fact that there is a user-activated behavior is controlled by assigning the "User.Activation" tag. We use an Eval script to assign the behaviors, wrapping them within a test of the activated ability state. This edge also demonstrates the need to carefully distinguish between using the "#traitbonus" and "#traitroll" macros. The net result is the definition below.

```
<thing
  id="edgBerserk"
  name="Berserk"
  compset="Edge"
  isunique="yes"
  description="Description goes here">
  <fieldval field="edgMinRank" value="0"/>
  <tag group="EdgeType" tag="Background"/>
```

```
  <tag group="User" tag="Activation"/>

 <eval index="1" phase="PreTraits" priority="5000">
   <before name="Calc trtFinal"/><![CDATA[
   if (field[abilActive].value = 0) then
     #traitbonus[trParry] -= 2
     #traitbonus[trTough] += 2
     #traitroll[skFighting] += 2
     endif
   ]]></eval>
 </thing>
```

ARCANE BACKGROUNDS

Each arcane background is granted via an edge. That means that each arcane background thing must automatically add all of the proper logic for managing that arcane background. One thing can automatically add any number of chained things by use of the "bootstrap" mechanism within the Kit, so our edge will bootstrap the logic for the arcane background. The "Magic" background below demonstrates this.

```
<thing
  id="edgArcMag"
  name="Arcane Background: Magic"
  compset="Edge"
  isunique="yes"
  description="Description goes here">
  <fieldval field="edgMinRank" value="0"/>
  <tag group="EdgeType" tag="Background"/>
  <bootstrap thing="arcMagic"/>
  </thing>
```

TRAIT PRE-REQUISITES

Numerous edges possess pre-requisites on different traits, such as requiring a Strength attribute of d8 or higher, or needing a Fighting skill of d10 or higher. Each edge needs to implement these pre-requisites individually, but the structure and behavior is always the same. We use the "Sweep" edge as an example to show how this works below.

```
<thing
  id="edgSweep"
  name="Sweep"
  compset="Edge"
  isunique="yes"
  description="Description goes here">
  <fieldval field="edgMinRank" value="0"/>
  <tag group="EdgeType" tag="Combat"/>
  <exprreq message="Strength d8 required."><![CDATA[
   #trait[attrStr] >= 4
   ]]></exprreq>
  <exprreq message="Fighting d8 required."><![CDATA[
   #traitfound[skFighting] >= 4
   ]]></exprreq>
  </thing>
```

Note that each separate requirement is implemented as a distinct pre-requisite instead of merging them into a single test. While this is not required, it is a highly recommended approach. The advantage of doing this is that HL can readily report each individual requirement that is not satisfied, providing greater detail to the user about what is wrong and needs to be fixed.

PRE-REQUISITES ON OTHER EDGES

There are quite a few edges that are boosted versions of other edges. They can only be taken when the lesser version of the edge has already been taken. For example, the "Improved Arcane Resistance" edge can only be taken if the "Arcane Resistance" edge has already been taken. So we need to setup appropriate pre-requisites on the presence of other edges. This can be handled easily via use of the "pickreq" element. The implementation of "Improved Arcane Resistance" shown below demonstrates how this is done.

```
<thing
  id="edgArcRes2"
  name="Improved Arcane Resistance"
  compset="Edge"
  isunique="yes"
  description="Description goes here">
  <fieldval field="edgMinRank" value="0"/>
  <tag group="EdgeType" tag="Background"/>
  <pickreq thing="edgArcRes"/>
  </thing>
```

SPECIALIZED PRE-REQUISITES

A small number of edges employ very specialized pre-requisites. This entails using the generalized "prereq" element and a Validate script to perform the appropriate tests. An example of this is the "Power Surge" edge, which requires the character to possess an arcane skill with a rating of d10 or higher. The implementation of this edge is shown below.

```
<thing
  id="edgPwrSurg"
  name="Power Surge"
  compset="Edge"
  isunique="yes"
  description="Description goes here">
  <fieldval field="rnkMinRank" value="1"/>
  <fieldval field="edgIsWild" value="1"/>
  <tag group="EdgeType" tag="Wildcard"/>

 <prereq iserror="yes" message="Arcane Skill of d10
required.">
   <validate><![CDATA[
    ~go through all arcane skills and check for d10
    foreach pick in hero where "component.Skill & Arcane.?"
     if (eachpick.field[trtFinal].value >= 5) then
      @valid = 1
      done
      endif
     nexteach
    altthing.linkvalid = 0
    ]]></validate>
  </prereq>
 </thing>
```

DEFERRING COMPLEX SPECIAL CASES

There will be times when you run into special cases that you want to implement correctly but that will require a non-trivial amount of extra work. In those cases, the best strategy is to figure out a suitable interim solution and plan to implement a "proper" solution once everything else is in place. There are two shining examples of this among the edges of Savage Worlds.

The first is the "Scholar" edge, which requires the user to select two "Knowledge" skills at d8 or higher and confers a "+2" bonus to each of the selected skills. Allowing the user to make the selections requires that edges support two separate menus from which the user

can select one skill each. The presented list of skills must be filtered to only show the existing "Knowledge" skills, and it should ideally only include skills with a d8 rating or higher. This represents a fair amount of work to implement, so we're going to defer adding this until later. For the time being, we'll setup the "Scholar" edge to simply require that at least two "Knowledge" exist at d8 or higher. We'll also leave it up to the user to add two suitable permanent adjustments to apply the appropriate bonuses to the desired skills. While this solution is a bit clunky, it allows the user to achieve the desired result without requiring lots of extra development work - a perfect interim solution when you're trying to get your initial set of files operational.

The second example of complexity that is best deferred is the "Professional" edge, along with its siblings "Expert" and "Master". Each of these requires that the user select a trait that is at a d12 rating, with that trait being granted a "+1" bonus. Just like the "Scholar" edge, a menu is needed that properly filters the list of traits shown to those that qualify and from which the user can make the selection. As an interim solution, these three edges can simply verify that at least one trait is at a d12 rating and the user can add a permanent adjustment to apply the bonus to the appropriate trait.

Acknowledging Our Limitations

There is one edge that we simply can't implement with 100% completeness, unless we want to inconvenience the user. The "Gadgeteer" edge requires that the character have at least two "Knowledge" skills at a d6 or higher, and those skills must be scientific in nature. The problem is that the user is free to type in whatever he wants for the domain of the skill, so we have no way to determine whether a particular skill is scientific or not. In situations like this, we are forced to either add some contrived mechanism to get the information we need or defer the details to the user. Deferring to the user is almost always the best path to choose. With "Gadgeteer", we can verify that there are at least two "Knowledge" skills at a d6 or higher, but we must leave it up to the user to verify that both of them are truly scientific in focus.

Adding and Revising Initial Panels

Revise Configuration Form (Savage)

All of the most fundamental mechanics for the game system should now be in place, so we ought to start testing everything out and adapting the visual behaviors to give us access to everything we've added. Since each character starts out on the Configuration form, that's usually the best place to begin when adapting the visuals. You'll find the default contents of the Configuration form in the file "form_config.dat".

What Can Users Customize?

The fundamental question that needs to be asked when looking at the Configuration form is: "What can users customize?" Load the Savage Worlds data files and take a look at the current Configuration form. The hero and player names at the top seem reasonable. Showing the list of enabled configuration settings on the right also seems reasonable. So we simply need to focus our attention on the various starting resources visible on the left.

If we look at our revised "Actor" component and compare that to the options currently available to the user, the starting character points and ability slots are not applicable in Savage Worlds. In addition, there are a number of aspects of the character that should probably be exposed to the user. These include the starting cash, starting experience, and whether the character is a "wild card". We could potentially add the starting attribute and skill points to the list, but those are defined as fixed values for the game system, so it's probably best to manage any changes to those values via permanent adjustments on the "Personal" tab (which we'll deal with later).

Adding New Portals

There are two new pieces of data that we need to allow the user to specify: the starting experience and whether the character is a wild card. This new data needs to be integrated into the "cnfStart" template that is already in place. This template contains all of the portals on the left that we want to manipulate, so we'll add our new portals there.

We'll start by adding the starting experience. This requires two separate portals that work like the starting cash. One portal is the label that tells the user what the data is for and the other is an edit area where the user can type in the number to be used. The edit portal must be hooked up to the appropriate field within the "Actor" component to store the user-entered value, and we can qualify the behavior to limit it to a maximum of three characters and accept only integers. The net result is two portals that look something like those below.

```
<portal
  id="lblxp"
  style="lblNormal">
  <label_literal
    text="Starting XP:">
    </label_literal>
  </portal>

<portal
  id="xp"
  style="editCenter">
  <edit     field="acStartXP"
    maxlength="3"
    format="integer">
    </edit>
  </portal>
```

We also need to let the user specify whether the character is a wild card. Since this is an either-or state, we can use a variety of methods, but we're going to use a checkbox for this example. The checkbox needs to be associated with the appropriate field within the "Actor" component where the selected state is stored, and it can display text adjacent to the check area to tell the user what it's for. We can also assign "tiptext" that appears when the user pauses the mouse over the portal. The end result is shown below.

```
<portal
  id="iswild"
  style="chkNormal"
  tiptext="Check this option to designate the character as a Wild
Card">
  <checkbox
    field="acIsWild"
    message="Wild Card Character?">
    </checkbox>
  </portal>
```

Positioning Portals

Once the portals have been added, they all start with default dimensions and a position of (0,0), which places them in the upper left corner of the template. That's not going to do us any good,

since we need to position them beneath the current portals. For positioning the starting experience, we can clone the same logic that is already used for positioning the starting cash. All we need to do is change the portal ids to operate on the "lblxp" and "xp" portals instead of "lblcash" and "cash", then position the "xp" portal relative to the "cash" portal instead of "menuabil".

Since we only have a single portal for the wild card designation, things are even simpler. The checkbox portal is automatically sized to occupy the space it needs, so we can simply center it horizontally. Then all we need to do is vertically position it beneath the "xp" portal, using the same technique as we did for the "xp" portal relative to the "cash" portal. The net result is the following lines of script code that should be added:

```
~position the starting xp beneath the starting cash
 perform portal[xp].alignrel[ttob,cash,15]
 perform portal[lblxp].centeron[vert,xp]
 portal[xp].width = 50
 portal[lblxp].left = (width - portal[lblxp].width - portal[xp].width -
10) / 2
 perform portal[xp].alignrel[ltor,lblxp,10]

~position the wildcard checkbox beneath the starting xp
 perform portal[iswild].centerhorz
 perform portal[iswild].alignrel[ttob,xp,15]
```

The final adjustment is to the overall height of the template. We need to ensure that the template height extends to the bottom of the bottommost portals, and the bottom portal is now the "iswild" portal. We also need to include a little bit of extra padding space to allow for gaps around elements. So change the last line of the script as shown below and our changes should be ready to test out.

```
height = portal[iswild].bottom + 3
```

To test your changes, start by using Quick-Reload to reload them into HL, then display the Configuration form via the "Character" menu. You should see the two new pieces of information ready to be edited. If you enter new values, close the form, and then re- open it, the changes should be retained. They are being saved in the appropriate fields within the "actor" pick that is automatically added to every new character.

DELETE OLD PORTALS

We decided above that we weren't going to be keeping the two menus that came pre-built into the Configuration form, so let's get rid of them now. The first step is to physically delete the two menu portals from the template. Next, we need to delete references to them from the Position script. This amounts to deleting the two sets of lines focused on positioning the menus and modifying the starting cash to be vertically positioned relative to the "label" portal instead of the "menuabil" portal. Since we keep the positioning of each portal grouped together, removing portals is a very simple process.

PRUNE EXTRANEOUS MATERIAL (SAVAGE)

Now is an excellent time to do a little cleaning up of our data files. This is something that you should get in the habit of doing on an ongoing basis as you develop your data files. The Skeleton data files contain pieces that you won't need for your game system, and these need to be disposed of. It's often easier to do it along the way as opposed to leaving everything for one major cleanup at the end. You'll still have some final cleanup to do, but it won't be as much work if you do some of it along the way. In addition, if you don't clean things up, your data files will be unnecessarily harder to maintain, so we recommend you keep your data files cleaned up.

DELETE EXTRANEOUS FILES

The Skeleton data files included the "thing_abilities.dat" file, but we won't be using it anywhere within the Savage Worlds files. So we can simply delete it.

DELETE ABANDONED MECHANICS

Take a look at the various core mechanisms that were provided by the Skeleton files and see if there are fields, scripts, things, tags, or anything else that won't be needed for your game system. If you find something, you can delete it. However, we recommend that you don't actually delete something immediately. Instead, try commenting it out and re-compiling the data files. This will uncover all of the places that rely on the item you want to delete. You can continue this process of commenting out pieces until the data files re-compile, re-load, and work exactly as they did before. Once you know that you can safely delete something, so can then do so. By using this technique, you'll avoid inadvertently deleting something that has dependency tendrils into other places that will be difficult to untangle, deferring such cases until later.

Let's start by deleting a few things. It's easiest to delete things, since they are usually the least likely to have other mechanisms that are dependent upon them. The Skeleton files include a number of derived traits that are not used within Savage Worlds, so we should delete them. Open the file "thing_traits.dat" and locate the "Resistance" trait. Comment it out, then re-compile the data files. Everything works fine, so we know nothing is dependent on the trait, and it can be properly deleted. Now try commenting out the "Power Points" traits and re-compiling. Errors are reported that you can chase down now, or you can defer this thing until later. It's easiest to defer it until later, so remove the comments and restore the thing. The same problem applies to the "Defense" and "Initiative" derived traits, so just leave them alone for now and we'll deal with them later.

Looking at the "Actor" component, there are two fields that are of no use for Savage Worlds: the starting character points and starting ability slots. Commenting them out and re-compiling uncovers that there are two resources that rely on the fields. Commenting out the two resources and re-compiling reveals that there are lots of places where the resources are referenced. So we should probably put everything back the way it was and hold off deleting these fields until later - after we've had a chance to re-work more of the data files for Savage Worlds and eliminate those existing dependencies.

Aside from the fields and resources identified above, a quick scan doesn't reveal anything that is clearly unused. There are some facets of equipment that will be different for Savage Worlds, but we haven't converted any of that yet, so it's premature to start cleaning that up. We'll leave things as they are for now and do further cleanup passes after we get more of the conversion completed.

A TAB FOR SKILLS (SAVAGE)

In Savage Worlds, skills are a bit more complex that within the Skeleton files. For example, skills like "Knowledge" require that the user be able to specify a suitable domain. This requires more horizontal space than the half-width provided on the "Basics" panel.

So we need to move skills to their own tab. This section walks you through the process.

RE-PURPOSE EXISTING TAB

The Skeleton files provide an "Abilities" tab that we don't need for Savage Worlds, so we'll re-purpose that tab as the new "Skills" tab. We start by renaming the file "tab_abilities.dat" to "tab_skills.dat". Then we need to open the file "tab_skills.dat" and convert the contents over to behave as a "Skills" tab.

At the bottom of the file, the "abilities" panel is defined. Change the id to "skills" and the name to "Skills". The Live tag expression must be changed to key on a "skills" tag (that we'll define in just a moment). And the layout referenced should be "skills" instead of "abilities".

Just above the panel definition is the layout definition. Change the id from "abilities" to "skills". The portal referenced should be changed to "skSkills". This entails that uses of the portal id within the Position script must also be changed to "skSkills" (there are three of them).

Moving to the top of the file, there is the table portal that needs to have its unique id changed to "skSkills". Within this portal, a variety of other facets must also be changed. The component must be "Skill" to show skills. The "showtemplate" should be changed to "skPick", and the "addthing" should be changed to "resSkill". Within the various scripts for the portal, change references from "Abilities" to "Skills" and uses of the "resAbility" resource to "resSkill".

Beneath the table portal is the template that needs to have its unique id changed to "skPick". The component set must be changed to "Skill" and the name should be changed to something appropriate. All other facets of the template can remain unchanged for now, as they are generic and will work fine for skills.

Once all these changes have been made, you should end up with something that looks like the following:

```
<portal
  id="skSkills"
  style="tblNormal">
  <table_dynamic
    component="Skill"
    showtemplate="skPick"
    choosetemplate="SimpleItem"
    fixedlast="yes"
    addpick="resSkill">
    <candidate>!Hide.Skill</candidate>
    <titlebar><![CDATA[
      @text = "Add a Special Skill - " &
hero.child[resSkill].field[resSummary].text
      ]]></titlebar>
    <headertitle><![CDATA[
      @text = "Skills: " &
hero.child[resSkill].field[resSummary].text
      ]]></headertitle>
    <additem><![CDATA[
      ~if we're in advancement mode, we've been frozen, so
display accordingly
      if (state.iscreate = 0) then
        @text = "{text a0a0a0}Add/Increase Skills Via Advances
Tab"
        done
        endif

      ~get the color-highlighted "add" text
      @text = field[resAddItem].text
      </table_dynamic>
  </portal>

<template
  id="skPick"
  name="Skill Pick"
  compset="Skill"
  marginhorz="3"
  marginvert="2">

  <portal
    id="name"
    style="lblNormal"
    showinvalid="yes">
    <label
      field="name">
      </label>
    </portal>

  <portal
    id="info"
    style="actInfo">
    <action
      action="info">
      </action>
    <mouseinfo mousepos="middle+above"/>
    </portal>

  <portal
    id="delete"
    style="actDelete"
    tiptext="Click to delete this item">
    <action
      action="delete">
      </action>
    </portal>

  <position><![CDATA[
    ~set up our height based on our tallest portal
    height = portal[info].height

    ~if this is a "sizing" calculation, we're done
    if (issizing <> 0) then
      done
      endif

    ~position our tallest portal at the top
    portal[info].top = 0

    ~center the other portals vertically
    perform portal[name].centervert
    perform portal[delete].centervert

    ~position the delete portal on the far right
    perform portal[delete].alignedge[right,0]

    ~position the info portal to the left of the delete button
    perform portal[info].alignrel[rtol,delete,-8]

    ~position the name on the left and let it use all available space
    portal[name].left = 0
    portal[name].width =
minimum(portal[name].width,portal[info].left - 5)

    ~if the ability is auto-added, change its font to indicate that
fact
    if (candelete = 0) then
      perform portal[name].setstyle[lblAuto]
      endif
    ]]></position>
  </template>

<layout   id="skills">
  <portalref portal="skSkills" taborder="10"/>
  <position><![CDATA[
```

```
    ~freeze our table in advancement mode to disable adding
new choices
    ~Note: All freezing must be done *before* any positioning is
performed.
    if (state.iscreate = 0) then
      portal[skSkills].freeze = 1
      endif

   ~position and size the table to span the full layout; it will only
use the
    ~vertical space that it actually needs
     perform portal[abAbility].autoplace
     ]]></position>
   </layout>

 <panel
   id="skills"
   name="Skills"
   marginhorz="5"
   marginvert="5"
   order="120">
   <live>!HideTab.skills</live>
   <layoutref layout="skills"/>
   <position><![CDATA[
    ]]></position>
   </panel>
```

Getting Everything to Work

If we were to re-compile the data files at this point, we would have an error that still needs to be resolved, so we'll address it now. Open the "tags.1st" file and locate the "HideTab" tag group. The "abilities" tag needs to be changed to "skills", since that's what we now key on within the Live tag expression of the panel.

There are also a few panel linkages on components that are currently tied to the "abilities" tab as a placeholder. These linkages are for edges and the like that have no relationship to the "Skills" tab, but we need to change them to something so the compiler can resolve everything. We can change the linkages to "skills", "basics", or any other tab, as long as the tab exists. We'll change it to "basics" as something safe for the time being. We'll fix it once we have a suitable tab in place to hook these components up to.

We should now be able to re-load our data files and see our new "Skills" tab appear. It isn't working correctly yet, but we'll take care of that in the following sections.

Internal Mechanics Changes

We now need to make some internal changes to mechanics in order to get skills behaving properly. The first thing to address is the fact that all skills are being automatically added to every character. In Savage Worlds, skills are purchased, so each character should start out with zero skills. Open the file "bootstrap.1st" and look for the various "enmasse" entries. Each of these entries automatically adds all things that satisfy a particular tag expression. One entry specifies all things with the "component.Skill" tag, which will add all skills. Delete this "enmasse" entry.

The next detail we need to address is that there may be some skills that the user is not allowed to purchase directly and that are automatically added (e.g. via edges, powers, etc.). This means that we need a method for designating some skills as hidden from the user for selection. A general mechanism for hiding things/picks is already defined via the "Hide" tag group, and we already have a tag for hiding skills in the "Hide.Skill" tag. We can assign this tag to any things that should be hidden from the user. The Candidate tag expression of the table portal will preclude such things from being shown for selection.

If errors are identified with any skills, it would be helpful to the user for the "Skills" tab to appear in red, drawing attention to the problem for easy correction. To simplify this, we need to associate all skills with the new tab. This can be accomplished by linking the "Skill" component to the new "skills" panel via a "panellink" attribute.

```
 <component
   id="Skill"
   name="Skill"
   autocompset="no"
   panellink="skills">
   ...
   </component>
```

The final detail is to eliminate the special validation thing that is defined for abilities. You'll find this thing in the file "thing_validate.dat" and it will have the unique id "valAbility". Since skills are governed by points, we don't need anything equivalent for Savage Worlds and can simply delete the thing.

Enhancing the Skill Template

The current "skPick" template within the file "tab_skills.dat" contains only generic mechanisms. We now need to enhance the template to include the important facets of skills that are currently missing. The first thing we need to add is an incrementer that will allow the user to adjust the die type for a given skill. We'll use the same incrementer style that is already being used for attributes. So we add the portal shown below.

```
 <portal
   id="value"
   style="incrSimple">
   <incrementer
    field="trtUser">
    </incrementer>
   <mouseinfo mousepos="middle+above"><![CDATA[
    if (hero.tagis[mode.creation] = 0) then
      @text = "Skills must be modified via the Advances tab once
the character is locked for  play."
    elseif (autonomous = 0) then
      @text = "This trait has been improved via the Advances tab
and cannot be modified further  from here."
    else
      @text = "Allocate points to this skill by clicking on the
arrows to increase/decrease the number of points assigned."
      endif
    ]]></mouseinfo>
   </portal>
```

We also need to allow the user to specify a suitable domain for skills that require one, such as the "Knowledge" skill. We need an edit portal in order to let the user enter the name of the domain, plus we need a label portal to identify the empty edit portal as being for the domain. We want the label portal to be in a smaller font and softer color than the name of the skill itself, so we choose a suitable style that reflects that behavior. The result is that we add the two portals shown below.

```
 <portal
   id="lbldomain"
   style="lblSecond">
   <label
    text="Domain:">
```

```
    </label>
  </portal>

<portal
  id="domain"
  style="editNormal">
  <edit
    field="domDomain">
    </edit>
  </portal>
```

Now that the portals have been added to the template, we need to properly position everything. We want to put the incrementer on the far left, with the name of the skill just to the right of the incrementer. If the skill requires a domain, we'll place it just to the right of the skill name, but we need to hide the portals associated with the domain if the skill does not require a domain. By default, all of the portals are always shown for every skill, so we control the visibility of each portal to ensure the contents being displayed reflect the correct behaviors for each skill. We can determine whether a skill needs a domain based on the presence of the "User.NeedDomain" tag. Putting it all together yields the following new Position script.

```
~set up our height based on our tallest portal
 height = portal[info].height

~if this is a "sizing" calculation, we're done
 if (issizing <> 0) then
   done
   endif

~position our tallest portal at the top
 portal[info].top = 0

~position the other portals vertically
 perform portal[name].centervert
 perform portal[domain].centervert
 perform portal[lbldomain].centervert
 perform portal[value].centervert
 perform portal[delete].centervert

~position the delete portal on the far right
 perform portal[delete].alignedge[right,0]

~position the info portal to the left of the delete button
 perform portal[info].alignrel[rtol,delete,-8]

~position the incrementer on the left
 portal[value].left = 0

~position the name next to the incrementer
 perform portal[name].alignrel[ltor,value,10]

 ~if we don't need a domain, hide it and let the name use all
available space
  if (tagis[User.NeedDomain] = 0) then
    portal[lbldomain].visible = 0
    portal[domain].visible = 0
    portal[name].width =
minimum(portal[name].width,portal[info].left - portal[name].left -
10)

 ~otherwise, position the domain portals next to the name
  else
    perform portal[lbldomain].alignrel[ltor,name,20]
    perform portal[domain].alignrel[ltor,lbldomain,5]
    portal[domain].width = 150
    endif

 ~if the ability is auto-added, change its font to indicate that fact
    perform portal[name].setstyle[lblAuto]
    endif
```

PROBLEM WITH DOMAINS

Unfortunately, there is a problem with the way we're handling domains. The "Domain" component automatically integrates the domain into the name of the pick. This means that entering a domain of "foo" for a "Knowledge" skill will change the name shown for the pick to "Knowledge: foo". Since we are also showing an edit portal for the domain, this behavior will appear wrong to the user.

We need to show a name for the pick that excludes the domain. For that, we'll need to utilize the basic name for the pick. That name is accessible via the "thingname" field, so we need to change the "name" portal to reference the "thingname" field instead of the full name. Once we do this, everything works nicely.

DELETE OLD SKILLS TABLE

The final task we need to complete is to delete the old "Skills" table from the "Basics" tab. Open the file "tab_basics.dat" and locate the "baSkill" table portal. This table of skills is no longer applicable, so delete it. Now scroll further down to the "baSklPick" template. This template is only used by the table portal we just deleted, so it can also be deleted. Finally, we need to delete the table from the "basics" layout. This amounts to deleting the appropriate "portalref" element and deleting the positioning details for the portal from the Position script.

NOTE! Remember that you can always comment out sections instead of deleting them until you're certain what can be safely deleted.

REVISE "BASICS" TAB (SAVAGE)

Now would be an excellent time for us to revise the "Basics" tab to contain the material we want it to show. In general, that material consists of the proper set of derived traits for Savage Worlds and details relating to the creation and advancement of the character, such as rank, XP, and unallocated resources.

DERIVED TRAITS

The first thing we'll adjust visually is move the derived traits list up to the right instead of below the attributes. Derived traits are important and it makes sense to feature them prominently on the tab, so we'll put them next to the attributes. We'll start by moving the table portal within the Position script of the layout. This is as simple as revising the script to position the portal as shown below.

```
portal[baTrait].width = portal[baAttrib].width
portal[baTrait].left = width - portal[baTrait].width
portal[baTrait].height = height
```

We now need to prune the list to only show the derived traits that we want for Savage Worlds. One option is to actually delete all the extraneous derived traits, but they are hooked into the Skeleton logic and will require some work to eliminate, so we'll simply hide them for the time being and eliminate them fully later one. The standard technique for this type of approach is to define a new tag in the "Hide" group, assign it to traits that we don't want shown, and then ignore those within the table. That's the technique we'll use. We start by adding a "Trait" tag to the "Hide" group, then we

assign that tag to all derived traits that were inherited from the Skeleton files.

```
<tag group="Hide" tag="Trait"/>
```

Next, we modify the "baTrait" table portal to have a List tag expression that excludes all traits with the "Hide.Trait" tag, as shown below. After that, we're good to go.

```
<list>!Hide.Trait</list>
```

Once the list is pruned, we'll make sure the remaining traits are shown in the order that they are presented on the Savage Worlds character sheet, since that will provide a consistent and familiar organization for the user. To manually sequence things in a table, HL provides the built-in "explicit" tag group and "explicit" sortset. The sortset is already employed by the table portal, so we just need to make sure each thing is properly configured. The "explicit" tag group is "dynamic", which means we can define tags implicitly by just assigning whatever tags we need to things. The tag group also presumes that all tags are numeric values. So we can assign an "explicit.1" tag to the "Pace" trait, an "explicit.2" tag to the "Parry" trait, etc.

The derived traits table just lists the traits now, and it should really have a title above it. We can include one by adding a "headertitle" element to the table portal, specifying the text we want displayed. The new element should look like the one shown below.

```
<headertitle><![CDATA[
  @text = "Derived Traits"
  ]]></headertitle>
```

The last thing we should do is change the visual presentation of derived traits slightly - i.e. by making them a little bigger and reducing the gap between the names and the values. We can reduce the spacing by increasing the outer margins on the left and right of the template. This can be accomplished by changing the "marginhorz" attribute to a value like "30". We can increase the size of the text shown slightly by changing the style associated with the "name" and "details" portals from "lblNormal" to "lblLarge". We can also improve the spacing and relative placement of the portals by allowing the "name" portal to auto-size itself and assigning a fixed width to the "details" portal, since it's just a simple numeric value. This results in the last two sections of the Position script being changed to something like the following:

```
~position the name on the left
 portal[name].left = 0

~position the details to the left of the info portal
 portal[details].width = 40
 perform portal[details].alignrel[rtol,info,-10]
```

CHARACTER RANK

The next thing we should do with the "Basics" tab is add details about the character's rank and accrued experience points, and we should probably put it beneath the derived traits. The tab currently has a "Horizontal" portal that is beneath the attributes, so we'll start by moving it over to beneath the derived traits table portal. This can be done by simply changing the Position script for the layout. The pertinent lines for placing the separator are shown below.

```
~position the separator beneath the derived traits
 portal[Horizontal].top = portal[baTrait].bottom + 15
 portal[Horizontal].left = portal[baTrait].left +
(portal[baTrait].width - portal[Horizontal].width)   / 2
```

Displaying the character's rank and accrued XP entails showing some text for a single facet of the character. Consequently, it only requires a simple label portal, as opposed to the tables we've been using thus far. We'll define our new label portal outside of a template, just like we've been defining the table portals. Labels don't need to be within a template unless they are linked to fields, so we'll use a script-based label and avoid needing the extra overhead of a template. The script will display the character rank and put the current XP in parentheses, with the entire portal looking something like the one shown below.

```
<portal
  id="baRank"    style="lblLarge">
  <label>
    <labeltext><![CDATA[
      var rank as number
      rank = herofield[acRank].value
      if (rank = 0) then
       @text = "Novice"
      elseif (rank = 1) then
       @text = "Seasoned"
      elseif (rank = 2) then
       @text = "Veteran"
      elseif (rank = 3) then
       @text = "Heroic"
      else
       @text = "Legendary"
        endif
      @text &= " (" & hero.child[resXP].field[resMax].value & "
XP)"
      ]]></labeltext>
    </label>
  </portal>
```

Once the portal is defined, we next need to add it to the layout and position it properly. Adding it to the layout amounts to adding a "portalref" element that references our new portal. We also need to specify where to sequence the contents of the portal in the overall tab sequence within the layout. This is controlled via the "taborder" attribute. Then we need to properly position the portal beneath the separator within the Position script for the layout. Both the new reference and the additional script code are shown below.

```
<portalref portal="baRank" taborder="30"/>

~size and position the rank details table beneath the separator
 portal[baRank].width = portal[baTrait].width
 portal[baRank].left = portal[baTrait].left
 portal[baRank].top = portal[Horizontal].bottom + 15
```

ANOTHER SEPARATOR

We still want to display information about the character creation state to the user, and that should probably go beneath the character rank. However, it should also be clearly decoupled from the rank, so we need to add another separator to the layout that we'll place beneath the rank. The state information can then be placed beneath the second separator.

Every use of a portal via a "portalref" element assumes the official name of the portal is name to be used within the layout. However,

we want to use the same portal multiple times within a single layout. This requires that we give each portal reference a distinct "logical name" for use within the layout, which is accomplished via the "reference" attribute within the "portalref" element.

We'll start by changing the existing "Horizontal" portal to use a new reference id of "separator1". This requires we modify the "portalref" element as shown below and then change all references to "Horizontal" within the Position script to use "separator1" instead.

```
<portalref reference="separator1" portal="Horizontal"/>
```

We can now add the second separator. To do this, we'll duplicate the existing "portalref" element in the layout. Then we'll change the "reference" attribute to indicate a different name. We'll call it "separator2". The last thing we need to do is properly position the new separator beneath the character rank. That can be done by adding the following lines of code at the end of the Position script of the layout.

```
~position the second separator beneath the rank details
 portal[separator2].top = portal[baRank].bottom + 15
 portal[separator2].left = portal[baRank].left +
(portal[baRank].width - portal[separator2].width) /   2
```

Tracking Advances

The final piece we need to add to the "Basics" tab is the list of character creation details, but we can't do that quite yet. Most of the information we need to display is already available through a variety of resources, yet there is one piece of information that is currently missing. We need to track the number of "advances" that the character has accrued and applied. The number of accrued advances is calculated based on the number of experience points accrued (e.g. one advance per five XP). This is most easily handled as a new resource, so we'll define the resource we need in the file "thing_miscellaneous.dat". This resource will auto-calculate the quantity available based on the XP, resulting in something similar to what's presented below.

```
<thing
  id="resAdvance"
  name="Advances"
  compset="Resource">
  <fieldval field="resObject" value="Advance"/>
  <tag group="Helper" tag="Bootstrap"/>
  <eval index="1" phase="Setup" priority="2000">
    <after name="Calc XP Max"/><![CDATA[
    var xp as number
    xp = #resmax[resXP]
    if (xp < 80) then
      field[resMax].value = round(xp / 5,0,-1)
    else
      xp -= 80
      field[resMax].value = 16 + round(xp / 10,0,-1)
      endif
    ]]></eval>
  </thing>
```

Character Creation Details

All of the character creation details that we need to display are now being properly tracked via resources, so we can add a new table that lists the various resources and presents them to the user. In order to do this, we need to identify each resource to be included in the list. We define a new "Helper.Creation" tag for this purpose, then we assign the tag to all resources that pertain to character creation details. This includes advances, attribute points, skill points, hindrances points, and edges.

As we did for derived traits, we should specify the sequence in which the various character creation details are presented to the user. This entails assigning an "explicit" tag with a suitable numeric order value to each resource that we just assigned a "Helper.Creation" tag. Then we can use the "explicit" sortset to display all the details in the order we want. The revised "resHinder" resource is shown below.

```
<thing
  id="resHinder"
  name="Hindrance Points"
  compset="Resource">
  <fieldval field="resObject" value="Reward"/>
  <tag group="Helper" tag="Bootstrap"/>
  <tag group="Helper" tag="Creation"/>
  <tag group="explicit" tag="3"/>
  </thing>
```

We can now define a new table portal wherein we'll display the character creation details. We limit the list of picks included in the table to only include those with the "Helper.Creation" tag. The net result is something similar to the following:

```
<portal
  id="baCreation"
  style="tblInvis">
  <table_fixed
    component="Resource"
    template="baResource"
    sortset="explicit"
    scroller="no">
    <list>Helper.Creation</list>
    </table_fixed>
  </portal>
```

With the table portal in place, the next step is to define an appropriate template for displaying resources in the list. We can easily adapt the "baTrtPick" template for this purpose. We can start by copying the entire template. After that, we can systematically convert the template. We start by changing the unique id and eliminating the "info" portal. We then change the script that generates the text for display in the "details" portal. Resources already synthesize a suitable text summary that is perfect for our needs, so we simply need to adapt the script over to utilize the "resShort" field. We can revise our margins to something more suitable for our purposes. Lastly, we re-work the Position script to omit the "info" portal, position the "details" on the far right, and put the name on the left. When we're done, our new template should look similar to the one shown below.

```
<template
  id="baResource"
  name="Resource Pick"
  compset="Resource"
  marginhorz="25"
  marginvert="3">

  <portal
    id="name"
    style="lblLeft"
    showinvalid="yes">
    <label>
```

```
      <labeltext><![CDATA[ @text = field[name].text & ":"
]]></labeltext>
      </label>
    </portal>

  <portal
    id="details"
    style="lblLeft">
    <label>
      <labeltext><![CDATA[ @text = field[resShort].text
]]></labeltext>
      </label>
    <mouseinfo mousepos="middle+above"><![CDATA[
      @text = "???"
      ]]></mouseinfo>
    </portal>

  <position><![CDATA[
    ~set up our height based on our tallest portal
    height = portal[name].height

    ~if this is a "sizing" calculation, we're done
    if (issizing <> 0) then
      done
      endif

    ~center the portals vertically
    perform portal[name].centervert
    perform portal[details].centervert

    ~position the details portal on the far right
    portal[details].width = 55
    perform portal[details].alignedge[right,0]

    ~position the name on the left
    portal[name].left = 0
    ]]></position>
  </template>
```

The final task is to properly add the new portal to the layout and position it beneath the character rank and the lower separator. Adding the portal to the layout consists of adding a new "portalref" element, along with an appropriate tab order. Once added, it can be readily positioned at the end of the Position script for the layout. By adding the script code below at the bottom of the script, the new portal is conveniently positioned beneath the character rank and separator.

```
portal[baCreation].width = portal[baTrait].width
portal[baCreation].left = portal[baTrait].left
portal[baCreation].top = template[separator2].bottom + 15
portal[baCreation].height = height - portal[baCreation].top
```

INCREMENTER BEHAVIOR (SAVAGE)

One glaring facet of the interface is that the incrementers used for attributes and skills are too narrow. While there is enough space to show the die type, any roll adjustment gets cut off. In addition, the interactive behavior of the incrementers is problematic in one respect. So we need to address these aspects.

DEFINE A NEW STYLE

The Skeleton files provide an assortment of pre-defined styles that we can build upon. However, for Savage Worlds, we need a special style that is ideal for showing and manipulating the die types of attributes and skills. All of the styles that are used within the user-interface are defined in the file "styles_ui.aug", so open that file. Locate where the various styles are defined for incrementers - there are three of them pre-defined for us.

What we need is a customized version of the "incrSimple" style, so we'll start by copying that style. We need an appropriate id for our new style, so we'll use "incrDie" to clearly indicate that this style is intended for use in managing die types. The color and font details are fine, but we need to widen the incrementer a bit. In addition to increasing the full width of the incrementer, we also need to increase the area for showing text and we need to adjust the position where the "plus" behavior is positioned. This means we need to adjust three of the attributes: "fullwidth", "textwidth", and "plusx". All three values should be adjusted by the same amount, and we'll use 20 as a reasonable increase. Why so much? Because the style is used for skills, where we need to handle the display of "d12+2", and that requires about 20 extra pixels of width.

A convenient feature of incrementers is that clicking within the text region allows the user to directly edit the contents as a textual value. For simple numbers, this feature is quite handy. However, we're multiplying the value by two and potentially including a roll adjustment, so a value displayed as "d8+1" would show the value "4" when the user tries to edit it. That's going to be quite confusing to users, so we're better off disabling this feature for our incrementer. This is accomplished by specifying the "editable" attribute with a value of "no". Putting it all together yields an incrementer style that looks similar to the one below.

```
  <style
    id="incrDie">
    <style_incrementer
      textcolorid="clrnormal"
      font="fntincrsim"
      editable="no"
      textleft="13" texttop="0" textwidth="44" textheight="20"
      fullwidth="70" fullheight="20"
      plusup="incplusup" plusdown="incplusdn" plusoff="incplusof"
      plusx="59" plusy="0"
      minusup="incminusup" minusdown="incminusdn"
minusoff="incminusof"
      minusx="0" minusy="0">
      </style_incrementer>
    </style>
```

USE THE STYLE

Once our new style is defined, we need to put it to use. There are only two places where we need to let the user manipulate the die type via an incrementer. The first is on the "Basics" tab where we manipulate attributes, so open the file "tab_basics.dat" and locate the "baAttrPick" template. Change the style of the incrementer used within it to "incrDie". The other location is on the "Skills" tab, so open the file "tab_skills.dat", locate the incrementer used for skills, and change that style as well. Re-loading the data files should now result in more appropriate behaviors for manipulating die types.

ADD AN "EDGES" TAB (SAVAGE)

The next logical step in the evolution of the Savage Worlds data files is to add a tab for edges. Due to the close relationship between edges and hindrances, we'll also include hindrances on the same tab, as well as the rewards that will be selected to offset any chosen hindrances. However, we'll simply call the tab "Edges". The sections below outline the process of adding and tailoring this new tab.

Something Simple to Start With

We need to a create an entirely new tab, so one option would be to start from scratch. A faster approach is to find an existing tab that is extremely simple and adapt it for use as our starting point. The "Skills" tab is perfect for this purpose, so we'll use it.

We first copy the file "tab_skills.dat" to "tab_edges.dat". If we try to re-compile the data files now, we'll get lots of errors about duplicate unique ids and such. So we need to rename the contents of the new file, and we'll tailor everything to use "edge" instead of "skill". This entails renaming all unique ids and names, changing all references to components and component sets, modifying all uses of "addpick" and actual references to resources, etc.

We also need to add a new "HideTab" tag for the new "edges" tab and adjust the Live tag expression to use it. Lastly, the template for skills includes handling for domains and an incrementer for adjusting the die types, neither of which we need. So we must strip those portals out of the template and adjust the Position script accordingly. We could potentially switch to using the "SimpleItem" template, except that we plan to add special support later on for edges like "Scholar" and "Professional", so we need a separate template that we can customize.

When the conversion is done, we'll have a single portal, template, layout, and panel. All of them will be tailored for the display of edges, and our new tab will look just like the "Skills" tab, except that it will manage the selection of edges. The four pieces will look like those shown below.

```
 <portal
   id="edEdges"
   style="tblNormal">
   <table_dynamic
    component="Edge"
    showtemplate="edEdge"
    choosetemplate="SimpleItem"
    addthing="resEdge">
    <titlebar><![CDATA[
     @text = "Add an Edge - " &
hero.child[resEdge].field[resSummary].text
     ]]></titlebar>
    <headertitle><![CDATA[
     @text = "Edges - " &
hero.child[resEdge].field[resSummary].text
     ]]></headertitle>
    <additem><![CDATA[
     ~if we're in advancement mode, we've been frozen, so
display accordingly
     if (state.iscreate = 0) then
      @text = "{text a0a0a0}Add Edges Via Advances Tab"
      done
      endif

     ~get the color-highlighted "add" text
     @text = field[resAddItem].text
     ]]></additem>
   </table_dynamic>
  </portal>

 <template
   id="edEdge"
   name="Edge Pick"
   compset="Edge"
   marginhorz="3"
   marginvert="2">

  <portal
   id="name"
   style="lblNormal"
    <label
     field="name">
     </label>
    </portal>

  <portal
   id="info"
   style="actInfo">
   <action
    action="info">
    </action>
   <mouseinfo mousepos="middle+above"/>
   </portal>

  <portal
   id="delete"
   style="actDelete"
   tiptext="Click to delete this item">
   <action
    action="delete">
    </action>
   </portal>

  <position><![CDATA[
   ~set up our height based on our tallest portal
   height = portal[info].height


   ~if this is a "sizing" calculation, we're done
   if (issizing <> 0) then
    done
    endif

   ~position our tallest portal at the top
   portal[info].top = 0

   ~position the other portals vertically
   perform portal[name].centervert
   perform portal[delete].centervert

   ~position the delete portal on the far right
   perform portal[delete].alignedge[right,0]

   ~position the info portal to the left of the delete button
   perform portal[info].alignrel[rtol,delete,-8]

   ~position the name on the left and use all availble space
   portal[name].left = 0
   portal[name].width =
minimum(portal[name].width,portal[info].left - portal[name].left -
10)

   ~if the ability is auto-added, change its font to indicate that
fact
   if (candelete = 0) then
    perform portal[name].setstyle[lblAuto]
    endif
   ]]></position>
  </template>

 <layout
   id="edges">
   <portalref portal="edEdges" taborder="10"/>
   <position><![CDATA[
    ~freeze our tables in advancement mode to disable adding
new choices
    ~Note: All freezing must be done *before* any positioning is
performed.
    if (state.iscreate = 0) then
     portal[edEdges].freeze = 1
     endif

    ~position and size the table to span the full layout; it will only
use the
    ~vertical space that it actually needs
```

```
    perform portal[skSkills].autoplace
    ]]></position>
  </layout>

<panel
  id="edges"
  name="Edges"
  marginhorz="5"
  marginvert="5"
  order="130">
  <live>!HideTab.edges</live>
  <layoutref layout="edges"/>
  <position><![CDATA[
  ]]></position>
  </panel>
```

Refining Edges

Our edges table is now in place, but it's still a little bit rough in how it works, so we need to refine its behavior. If you click within the "Edges" table to add a new edge, you'll see all of the edges listed properly. Our pre-requisites appear to be handled correctly, with any failed pre-requisites being listed in the description region and appropriate edges being greyed out when the pre-requisites aren't satisfied. However, some of the names of edges are being cut off due to the horizontal space being too narrow. This can be easily fixed by specifying a width of our own choosing for the "thing" template. Unfortunately, we're using the "SimpleItem" template as our "choose" template, so changing the width within that template will change it everywhere the template is used, and that's not what we want.

One solution would be to copy the "SimpleItem" template into the file "tab_edges.dat", rename it, and adapt it to our needs. That would work, but then we'd find ourselves copying the template whenever we wanted to make minor adjustments, and that will make the data files harder to maintain over time. Fortunately, the "SimpleItem" template has been designed so that it can be easily customized in situations like this. By assigning a "SimpleItem.widthXXX" tag to a thing, where "XXX" is the width value we want to use, the "SimpleItem" template will detect a custom width and use it. For example, a tag of "SimpleItem.width250" indicates a width of 250 pixels should be used. Since the tag needs to be assigned to all things to ensure that it will always be recognized by the template, we can assign the tag to the component and all things derived from that component will inherit the tag. This means that all we need to do to specify a custom width for the "SimpleItem" template when showing edges for selection is add the tag as shown below to the "Edge" component within the file "traits.str".

```
<tag group="SimpleItem" tag="width250"/>
```

The next thing of note is that the edges are sorted alphabetically, but they are not grouped like they are within the rulebook. By default, all tables list their contents alphabetically unless we specify something else. This is accomplish via the use of a "sort set". Open the file "control.1st" and scroll towards the bottom, where you'll find a number of "sortset" elements. These elements define the various sort sets that are provided by the Skeleton files. We need to add a new one that will sort the edges first by their type and then by name within each type. Each sort key within a sort set can be either a field or a tag group, specified by its unique id, and the order of the sort keys dictates the ultimate ordering of the items. We want our edges to be sorted by the type and then the name, so we'll need two sort keys. Putting it all together yields a sort set that looks similar to the one shown below.

```
<sortset
  id="Edge"
  name="Edge By Type and Name">
  <sortkey isfield="no" id="Edge"/>
  <sortkey isfield="no" id="_Name_"/>
  </sortset>
```

Once the sort set is defined, we can reference it from the table portal so that all edges are shown to the user in the order dictated by the sort set. This is accomplished by adding the "choosesortset" attribute to the "table_dynamic" element of the table portal and specifying the id of our new sort set. The new attribute should look like below.

```
choosesortset="Edge"
```

After making this change and experimenting with edges again, the behavior just doesn't seem very intuitive. While we've now listed the edges in the same organization as the rulebook, there is no clear structural association to the rulebook here. Coupled with the fact that there are some edge types with only a few edges and others with lengthy lists, this new ordering is likely to be more confusing than helpful to most users. Consequently, we're probably better off not using the new sort set and leaving the list in a purely alphabetical order.

This is the sort of trade-off that you'll need to make at times. Sometimes, it's better to emulate the rulebook's organization, while sometimes it's better to use a different approach that is better suited to the way users will be putting your data files to work.

Evolving the "Edges" Tab (Savage)

Once we have a basic tab in place that contains the "Edges" table, we need to add tables for selection of hindrances and rewards.

Adding Hindrances

We'll handle hindrances next, since we should place those immediately below the table of edges. We put them below, because the list of edges will evolve as the character advances, while the hindrances will generally be fixed after character creation.

Our first step is to add a table portal in which we can manage the hindrances. Since hindrances have a number of complexities we have to deal with, we will need our own custom template instead of being able to use the "SimpleItem" template. However, using the "SimpleItem" template on an interim basis makes it easy for us to get the table added, so that's what we'll do.

We copy the table portal for edges, then we adapt it for our needs. This amounts to changing all edge references over to hindrances. Hindrances don't have a resource associated with them, so we must also revise a few of the scripts slightly to eliminate references to the resource. The net result is the portal below.

```
<portal
  id="edHinders"
  style="tblNormal">
  <table_dynamic
    component="Hindrance"
```

```
      showtemplate="SimpleItem"
      choosetemplate="SimpleItem">
       <titlebar><![CDATA[
        @text = "Add a Hindrance"
       ]]></titlebar>
      <headertitle><![CDATA[
        @text = "Hindrances"
       ]]></headertitle>
       <additem><![CDATA[
       ~if we're in advancement mode, we've been frozen, so
display accordingly
        if (state.iscreate = 0) then
          @text = "{text a0a0a0}Cannot Add Hindrances After
Creation"
          done
          endif
       ~be sure to use a suitable color for a purely optional choice
       @text = "{text a0a0a0}Add New Hindrance"
       ]]></additem>
      </table_dynamic>
    </portal>
```

Our portal has now been added to the data files, but nothing is actually being done with it. We still need to integrate it into the layout and position it properly. Adding it to the layout is simple. We add a new "portalref" element that references the table portal. Since the table will be beneath the edges table, we'll assign a suitable tab order to ensure the flow proceeds from edges to hindrances. This results in the new element shown below.

```
<portalref portal="edHinders" taborder="20"/>
```

The next step is positioning the table portal properly. We'll reconcile the concerns of juggling three tables within the panel once they are all defined. For now, we'll simply position the new table beneath the table of edges. That can be achieved by adding the following lines of code to the Position script in the layout.

```
portal[edHinders].width = width
 portal[edHinders].top = portal[edEdges].bottom + 10
```

Customizing Hindrances

Now that the hindrances table portal is in place, we can customize how hindrances are displayed and behave. Although some hindrances are simple and involve only a name, there are some for which the user can selectively choose whether it is major or minor in impact. In addition, some hindrances need to specify a domain, just like skills. We need to provide our own template in order to support these behaviors.

Before we do anything, we need to get a new template that we can customize. We're currently using the "SimpleItem" template, and it would be an excellent starting point for our purposes. The template used for skills would also be an excellent starting point, so you could just as easily start with either one. We'll use the "SimpleItem" template, since that's what you will probably be using most often for this purpose. Open the file "visual.dat" and locate the "SimpleItem" template. Copy it into the file "tab_edges.dat", where we can adapt it.

Change the unique id to something suitable, such as "edHinder", and the component set to "Hindrance". Eliminate the extra logic in the Position script that is expressly for use when showing things, since we're always showing picks. Lastly, we need to hook the new template into the table portal, so go to the "edHinders" portal and change the "choosetemplate" to reference our new "edHinder" template. At this point, you should be able to re-compile and load, with no visible change in behavior.

We can now adapt the template to our needs. Many hindrances have a fixed severity rating, and we should be sure to display that clearly to the user. The name of each hindrance is currently shown as a field-based label that simply shows the value of the field. We'll change that to a script-based label, wherein we'll start with the name and append an indication of whether the hindrance is major or minor. If the hindrance has a user-controlled severity, we'll leave the name unchanged and show the state differently, as outlined further below. The net result is a revised "name" portal that should look something like the following.

```
<portal
  id="name"
  style="lblNormal"
  showinvalid="yes">
  <label>
   <labeltext><![CDATA[
    @text = field[name].text
    if (tagis[User.UserSelect] = 0) then
     if (field[hinMajor].value = 0) then
      @text &= " (Minor)"
     else
      @text &= " (Major)"
     endif
    endif
   ]]></labeltext>
  </label>
</portal>
```

For user-controlled hindrances, we need to add a new portal of some sort that allows the user to designate the severity. One option is to use a checkbox portal. A traditional checkbox would probably look quite poor, so we won't consider that further. However, we could also use a bitmap-based checkbox. With a bitmap-based checkbox, we select two separate bitmaps to indicate the two different states and rely on the bitmaps indicating the state to the user. This technique is used in a few places for different game systems and can be very useful. The problem with this approach is that we need to clearly convey "major" versus "minor" to the user via bitmaps - that's not an easy proposition.

Fortunately, we can also use a menu, just like we do for the character's gender on the "Personal" tab. We'll start by opening the file "tab_personal.dat", locating the "gender" portal, and copying it into the template we're working with so we can adapt it. We'll change the id to "severity" and hook it up to the "hinMajor" field. Then we can change the list of choices that the user is shown, making sure that the value zero reflects "minor" and one reflects "major", since that's the convention we've already established for the "hinMajor" field.

```
<portal
  id="severity"
  style="menuNormal">
  <menu_literal
   field="hinMajor">
   <choice value="0" display="Minor"/>
   <choice value="1" display="Major"/>
   </menu_literal>
  </portal>
```

The portal is added, so we now need to position it properly and control when it is made visible. We'll add the appropriate logic to the Position script for the template, keying on the presence of the "User.UserSelect" tag and positioning the severity to the right of the hindrance name. The additions made to the Position script should look similar to the code below.

```
~center the portal vertically
 perform portal[severity].centervert

~if we don't need a menu, hide the portal
 var edge as number
 if (tagis[User.UserSelect] = 0) then
   portal[severity].visible = 0
   edge = portal[name].right

~otherwise, position the menu portal to the right of the
name/domain
 else
   perform portal[severity].alignrel[ltor,name,15]
   portal[severity].width = 55
   edge = portal[severity].right
   endif
```

In addition to the severity menu, some portals also require a user-specified domain, just like certain skills. To support this, we need to add two portals that are virtually identical to the ones we used in the table of skills. So open the file "tab_skills.dat" and locate the "lbldomain" and "domain" portals within the "skPick" template. Copy them into the "edHinder" template that we're working on. The only aspect we need to change is the field associated with the "domain" edit portal, which needs to be hooked up to the "domDomain" field for hindrances. At this point, the two new portals should look similar to the ones below.

```
<portal
  id="lbldomain"
  style="lblSecond">
  <label
    text="Domain:">
    </label>
  </portal>

<portal
  id="domain"
  style="editNormal">
  <edit
    field="domDomain">
    </edit>
  </portal>
```

Once the portals are properly defined, we need to position them and control their visibility. If the domain is not needed, we simply hide it, else we position the domain to the right of the name - or the severity menu, if any. The new logic required in the Position script should look like below.

```
~center the portals vertically
 perform portal[domain].centervert
 perform portal[lbldomain].alignrel[btob,domain,0]

~if we don't need a domain, hide the portals
 if (tagis[User.NeedDomain] = 0) then
   portal[lbldomain].visible = 0
   portal[domain].visible = 0

~otherwise, position the domain portals next to the name,
making sure the
  else
    portal[lbldomain].left = edge + 15
    perform portal[domain].alignrel[ltor,lbldomain,5]
    portal[domain].width = minimum(150,portal[info].left -
portal[domain].left - 10)
    endif
```

With the integration of domain handling, we discover that we have the same problem we encountered previously for skills. The domain name is automatically synthesized into the name of the hindrance, which results in the domain being shown in the name and within the edit portal. We should omit it from the name. To accomplish that, we change the "name" portal to utilize the "thingname" field instead of the "name" field. Once that change is made, everything behaves as it should.

ADDING REWARDS

The one remaining table we need is for selecting rewards, which we'll place beneath hindrances. Rewards are very simple, consisting of only a name for display. As such, they don't need to take up the full width of the tab and we can display them in two columns instead of just one. We'll start by adding a table portal in which we can manage the rewards, and we'll configure it for two columns of display. Due to the simplicity of rewards, we can re-use the "SimpleItem" template and avoid having to define a new template. So we copy the table portal for edges and adapt it for our needs. This amounts to changing all references to edges over to rewards, as well as changing the edges resource over to the hindrance resource for tracking how many rewards are left to be selected by the user. Except for changing to two columns, everything else remains the same, resulting in the portal shown below.

```
<portal
  id="edRewards"
  style="tblNormal">
  <table_dynamic
    component="Reward"
    showtemplate="SimpleItem"
    choosetemplate="SimpleItem"
    addpick="resHinder"
    columns="2">
    <titlebar><![CDATA[
      @text = "Add an Reward - " &
hero.child[resHinder].field[resSummary].text
      ]]></titlebar>
    <headertitle><![CDATA[
      @text = "Rewards - " &
hero.child[resHinder].field[resSummary].text
      ]]></headertitle>
    <additem><![CDATA[
      ~if we're in advancement mode, we've been frozen, so
display accordingly
      if (state.iscreate = 0) then
        @text = "{text a0a0a0}Cannot Add Rewards After
Creation"
        done
        endif

      ~get the color-highlighted "add" text
      @text = field[resAddItem].text
      ]]></additem>
    </table_dynamic>
  </portal>
```

Just like we did for the hindrances table, we must now integrate the portal into the layout and position it properly. Adding it to the layout is achieved by adding a new "portalref" element that reference the table portal. Since rewards will be at the bottom, we'll

assign a tab order after the others. This results in the new element shown below.

```
<portalref portal="edRewards" taborder="30"/>
```

The final step is positioning the table portal properly. Just to get things working quickly, we position the rewards table beneath the table of hindrances by adding the following lines of code to the Position script in the layout.

```
portal[edRewards].width = width
 portal[edRewards].top = portal[edHinders].bottom + 10
```

Intelligent Positioning

We now have all three tables in place, but their positioning logic is incredibly crude. If enough edges are added, both the hindrances and rewards table could completely disappear beyond the bottom of the tab. So we need to change to logic that is more intelligent.

We start by establishing suitable minimum heights for the tables of hindrances and rewards, then calculating the vertical space remaining. Next, we allow the edges table to use as much vertical space as remains when those minimums are factored in. If the edge table doesn't need all the space, then the hindrances table is first allowed to expand as much as necessary. Lastly, if there is still unused space remaining, the rewards table is given an opportunity to expand. This logic ensures that we optimize our use of the vertical space, giving preference to edges and then hindrances, while we also ensure that the maximum amount of the edges table is kept visible at all times. The net result is the Position script shown below.

```
 ~freeze our tables in advancement mode to disable adding new
choices
  ~Note: All freezing must be done *before* any positioning is
performed.
  if (state.iscreate = 0) then
    portal[edEdges].freeze = 1
    portal[edHinders].freeze = 1
    portal[edRewards].freeze = 1
    endif

 ~determine the vertical gap we want to use between tables
  var gap as number
  gap = 15

 ~size all tables to span the full layout width
  portal[edEdges].width = width
  portal[edHinders].width = width
  portal[edRewards].width = width

 ~set the height of the small tables to be a maximum number of
rows for now
  portal[edHinders].maxrows = 3
  portal[edRewards].maxrows = 1

 ~position the rewards table at the bottom
  portal[edRewards].top = height - portal[edRewards].height

 ~position the hindrances table above the rewards
  portal[edHinders].top = portal[edRewards].top - gap -
portal[edHinders].height

 ~assign the remaining space to the edges table
  portal[edEdges].height = portal[edHinders].top - gap -
portal[edEdges].top
```

```
 ~position the hindrances table beneath the edges table and let it
  ~fill whatever vertical space is available between the edges
and rewards
  portal[edHinders].top = portal[edEdges].bottom + gap
  portal[edHinders].height = portal[edRewards].top - gap -
portal[edHinders].top

 ~position the rewards table beneath hindrance and let it expand
to fill space
  portal[edRewards].top = portal[edHinders].bottom + gap
  portal[edRewards].height = height - portal[edRewards].top
```

Reporting Errors

Earlier in the evolution process of our data files, we established panel linkages to the "basics" panel for edges, rewards, and hindrances. Consequently, if we add one of those three items to the character and the item is invalid in some way, the "Basics" panel will turn red to highlight the error to the user. Now that we have an appropriate panel in place for those three objects, we need to highlight the proper panel when errors are encountered. This is simply a matter of opening the file "traits.str" and changing the "panellink" attribute of the "Edge", "Hindrance", and "Reward" components. Change the attribute to associate the new "edges" panel with the component and everything will be hooked up properly.

Evolving Game System Elements

Character Creation Logic (Savage)

With all the basics now in place, it's a perfect time for us to get all of the basic character creation logic fully tested and operational. This entails tracking the allocation of attribute points and skill points, as well as the selection of races, edges, hindrances, and rewards.

Attribute Points

We'll focus on attribute points first. We already have a resource in place to track those points, so we need to properly consume those points. This can be most easily handled by adding a script to the "Attribute" component that spends the resource appropriately. The Skeleton files provide such a script for us already, so all we need to do is adapt it for our needs. The d4 rating for an attribute costs no points, but each increment beyond that costs one point. So we'll employ a component script that behaves accordingly, which should look similar to the one shown below.

```
 <eval index="2" phase="Traits" priority="10000">
   <before name="Calc resLeft"/>
   <after name="Bound trtUser"/><![CDATA[
   ~since the base starting value for each attribute is two, we
add only the extras
   perform #resspent[resAttrib,+,field[trtUser].value -
2,field[name].text]
   ]]></eval>
```

Skill Points

Skill points behave much like attributes, except that we have to treat the initial purchase of the skill as one skill point and all subsequent increases as either one or two additional skill points. We already have a resource in place to track the points, and the Skeleton files already provide a "Skill" component script that we can easily adapt to our needs.

The interesting logic here has to do with determining whether each die type for a skill costs one or two points. We can start by assuming that each notch counts at least one point. So we'll first calculate the total number of notches. We must also subtract one from our total, since the base values for skills is two, which costs the first point.

The extra skill point cost is incurred for each level that a skill exceeds its linked attribute. We can access the linked attribute and get its value via the "linkage" transition. Once we have that value, we can compare the two and add an extra skill point for each level of difference between them.

The net script that we need should look something like the following:

```
<eval index="2" phase="Traits" priority="10000">
  <before name="Calc resLeft"/>
  <after name="Bound trtUser"/><![CDATA[
  ~if this skill is not added directly to the hero (i.e. an advance),
skip it entirely
  if (origin.ishero = 0) then
    done
    endif

  ~the base value for skills is two, so we need to adjust by one
to get the proper cost
  var points as number
  points = field[trtUser].value - 1

  ~add an extra point for every level we exceed our linked
attribute
  var attrib as number
  attrib = linkage[attribute].field[trtUser].value
  if (field[trtUser].value > attrib) then
    points += field[trtUser].value – attrib
    endif

  ~accrue the total points spent on this skill
  hero.child[resSkill].field[resSpent].value += points
  ]]></eval>
```

RACES

Try choosing all of the different races for the character and verify whether everything is behaving correctly. This includes racial bonuses to attributes, skills, and derived traits, plus other facets such as the extra edge conferred by the "Human" and "Half-Elven" races. After a quick check, everything looks correct, except for one facet.

The issue is with the tallying of skill points. If the "Elven" race is selected, it confers a free bonus level to the "Agility" attribute. That bonus needs to be considered part of the base die type for the attribute. Otherwise, skills based on Agility will cost an extra skill point unless they are lower than the attribute.

Taking a look at the Eval script we created a few moments ago, we can easily accommodate this fix. The bonus die level tracked within the "trtBonus" field, so all we need to do is add the bonus to the user value and we'll have the net value we need. Note that we do not want to use the "trtFinal" field value, since that value also includes in-play adjustments, which are temporary and must be excluded from consideration here.

The code fragment below shows the effected lines within the Eval script and what they should now look like.

```
 ~get the level of our linked attribute and factor in bonuses due
to races and such
  var attrib as number
  attrib += linkage[attribute].field[trtBonus].value

 ~add an extra point for every level we exceed our linked
attribute
  if (field[trtUser].value > attrib) then
    points += field[trtUser].value – attrib
    endif
```

HINDRANCES

Add a variety of hindrances of different severities. Also add hindrances that have a user-controlled severity and toggle that severity. While doing so, watch the total points of rewards that are available for selection on the right. The total should go up one point for each minor hindrance and two points for each major hindrance. In addition, when hindrances are added that confer negative effects, verify that the effects are being applied properly. If a hindrance has user-controlled effects that can be toggled on and off, make sure the option appears properly on the "In-Play" tab and that it all behaves correctly.

Everything is working as it should, with one exception. The rules stipulate that a character may choose a maximum of one major hindrance and two minor hindrances, but this restriction is not being validated anywhere. We need to add a new validation rule that will enforce this restriction and report when a character exceeds the limits. To do that, we need to add a new thing whose sole purpose is performing this validation. By convention, we add these "validation" things to the file "thing_validate.dat". So open up the file and we'll add a new validation thing whose sole purpose is to invoke our validation rule to verify hindrances. We derive the thing from the "Simple" component set, and we must make sure it's added to every character so that the validation is always performed. The net result looks like the following.

```
<thing
  id="valHinders"
  name="Hindrances"
 compset="Simple">
  <tag group="Helper" tag="Bootstrap"/>
   <evalrule index="1" phase="Validate" priority="8000"
    message="Maximum of one major and two minor
hindrances allowed"
    summary="Limit exceeded"><![CDATA[
    ~iterate through all hindrances and tally up the number of
majors and minors
    var major as number
    var minor as number
    foreach pick in hero where "component.Hindrance"
      if (each.field[hinMajor].value = 0) then
        minor += 1
      else
        major += 1
        endif
      nexteach

    ~if we have no more than one major and two minor, we're
good
    if (major <= 1) then
      if (minor <= 2) then @valid = 1
        done
        endif
      endif

    ~mark associated tabs as invalid
     container.panelvalid[edges] = 0
     ]]></evalrule>
  </thing>
```

After making the above changes, validation is now working as it should. However, it would be better if the title above the hindrances table appeared in red to clearly identify where to fix the problem on the tab. This can be achieved by having the above EvalRule assign a tag to the hero, which can be checked by the table portal. So we start by defining a new "Hero.BadHinders" tag within the file "tags.1st". Add the following tag to the "Hero" tag group.

```
<value id="BadHinders"/>
 <!-- Indicates that the hindrances selected don't comply with
the rules  -->
```

Once that's in place, we can assign the tag within the above EvalRule script by adding the line below if the rule is failed.

```
perform hero.assign[Hero.BadHinders]
```

The final step is to check for and properly handle the tag within the table portal. Locate the "edHinders" portal within the file "tab_edges.dat" and then find the HeaderTitle script. This script returns the title to be displayed, which currently amounts to simply "Hindrances". We can change the script to detect if the tag is present and change the color to red when it is. The new script should look similar to the one below.

```
@text = ""
if (hero.tagis[Hero.BadHinders] <> 0) then
  @text = "{text ff0000}"
  endif
@text &= "Hindrances"
```

REWARDS

Add a variety of rewards and verify that the available number of resources is being properly adjusted. Some rewards are worth two points and some are worth one. Also verify that the effects of each reward are being applied correctly. Everything seems to check out fine.

EDGES

The final thing that we need to test on this tab is edges. Add a variety of edges and make sure that the appropriate resources are being adjusted. For edges that confer bonuses, verify that the effects are being applied properly. If an edge has user-controlled effects that can be toggled on and off, make sure the option appears properly on the "In-Play" tab and that it all behaves correctly. Once that's handled, we're ready to move on with our development.

REVISE ADJUSTMENTS (SAVAGE)

With all of the basic building blocks for characters now in place, we can assess the various temporary and permanent adjustments that can be applied to characters. The Skeleton files provide a starting set that we can convert appropriately and expand upon for Savage Worlds.

IDENTIFY NATURE AND CONTEXT

The first step is to identify all of the different adjustments that users may want to apply to a character. Once the nature of each adjustment is determined, then the context in which that adjustment can be applied must be assessed. Each adjustment must be designated as being applicable permanently, temporarily, or both. Some adjustments only make sense in one context or the other, such as attribute points only being a permanent change, while weapon attack bonuses only being a temporary change. However, there are quite a few that can be applied in either context.

In the case of Savage Worlds, there are many adjustments that can be applied to a character. After scanning the rulebook, the following set of adjustments should be supported, along with the valid context(s) for each.

- Attribute die type - permanent or temporary
- Attribute roll - permanent or temporary
- Skill die type - permanent or temporary
- Skill roll - permanent or temporary
- Derived trait - permanent or temporary
- Weapon attack - temporary only
- Number of advances authorized - permanent only
- Number of edges granted - permanent only
- Starting attribute points - permanent only
- Starting skill points - permanent only
- Number of arcane powers - permanent only
- Number of arcane power points - permanent or temporary
- Number of Bennies per game session - permanent

DEFINE ADJUSTMENTS

Once the list of adjustments is defined on paper, they can be implemented in the data files. Open the file "thing_adjustments.dat" and you'll find an assortment of adjustments provided by the Skeleton data files that we'll be modifying. There are a variety of tags and fields used by adjustments to appropriately configure the behavior of the adjustment. For each adjustment, the Eval script only applies its effects if the pick has been assigned the "Helper.Activated" tag. This tag is automatically assigned via a component script if the user activates the effects of adjustment. Permanent adjustments are implicitly activated, so their effects are always applied.

We'll examine two adjustments as examples. We'll start with the adjustment for the starting attribute points, since it doesn't involve the use of menu selection. The adjustment requires the user to manipulate its effects, so the "Helper.UserAdjust" tag is assigned. As part of that manipulation, an incrementer should be shown that allows the user to determine the modification to be applied, so the "AdjustShow.Increment" tag is assigned. Since attribute points only affect character creation, any adjustment to them is effectively permanent in nature, so we assign the "InPlay.PermOK" tag to indicate that this adjustment is suitable for selection as a permanent adjustment. The net result is something similar to the following.

```
<thing
  id="adjAttrPt"
  name="Attribute Points"
  compset="Adjustment"
  description="Select this adjustment to...">
  <tag group="AdjustShow" tag="Increment"/>
  <tag group="Helper" tag="UserAdjust"/>
  <tag group="InPlay" tag="PermOK"/>
  <eval phase="Setup" priority="8000">
    <before name="Calc trtFinal"/><![CDATA[
    if (tagis[Helper.Activated] <> 0) then
      #resmax[resAttrib] += field[adjUser].value
      endif
    ]]></eval>
  </thing>
```

Our second example is an adjustment for modifying the die type of an attribute. As with the example above, user manipulation is involved, so we use the "Helper.UserAdjust" tag. For manipulation, an incrementer is used to specify an amount and a menu is shown through which the user will select the attribute to be adjusted, so we have both the "Adjust.Increment" and "AdjustShow.Menu" tags. The behavior of the menu is controlled via two fields on the adjustment. The field "adjUsePick" dictates whether the menu should be selecting a thing or a pick, with a non-zero value indicating a pick. Since we want the user to select an attribute that is already added to the character, we specify a value of one. The field "adjCandid" specifies the tag expression to be used to identify the list of picks to display within the menu. Since we want to display all of the attributes, we use a value of "component.Attribute" for the field. Since an attribute adjustment might be either permanent or due to an in-game effort, we assign both tags to make it selectable as either a permanent or temporary effect.

The last detail of note is that the attribute to be adjusted is selected via the menu, so we need to identify the proper attribute through the menu. We assume that the menu portal associates the selection with the "adjChosen" field on the adjustment. We can then access the field and obtain the pick chosen within it via the "chosen." script transition. Once we have the proper attribute, we can then access fields on it using the standard methods and apply the adjustment appropriately.

Putting it all together yields an adjustment thing that looks very similar to the one shown below.

```
<thing
  id="adjAttrD"
  name="Attribute Die"
  compset="Adjustment"
  description="Select this adjustment to...">
  <fieldval field="adjUsePick" value="1"/>
  <fieldval field="adjCandid" value="component.Attribute"/>
  <tag group="AdjustShow" tag="Increment"/>
  <tag group="AdjustShow" tag="Menu"/>
  <tag group="Helper" tag="UserAdjust"/>
  <tag group="InPlay" tag="PermOK"/>
  <tag group="InPlay" tag="TempOK"/>
  <eval phase="Setup" priority="8000">
    <before name="Calc trtFinal"/><![CDATA[
    if (tagis[Helper.Activated] <> 0) then
      field[adjChosen].chosen.field[trtInPlay].value +=
field[adjUser].value
      endif
    ]]></eval>
  </thing>
```

VERIFY BEHAVIORS

The last step is to go through and verify that all of the adjustments work as we intend. When we do this, we'll discover that the template being used for selecting the adjustments is cutting off the edge of the name of long adjustments. If we open up the file "tab_personal.dat" and look for the table portal for adding adjustments, we'll see that the "choosetemplate" specified is "LargeItem". Similarly, if we open the file "tab_inplay.dat" and look for the table portal used to add adjustments, we'll see the same usage.

Since the "LargeItem" template is used for a variety of purposes, we can't modify the width without changing it for every use. However, the "LargeItem" template has been designed with the same customization logic as the "SimpleItem" template that we leveraged earlier. This means that we can define a suitable component tag on the "Adjustment" component to cause the "LargeItem" template to use a larger width when showing adjustments for selection by the user. All we need to do is add the XML element below to the "Adjustment" component and everything should look the way we want.

```
<tag group="SimpleItem" tag="width250"/>
```

MORE ON ARCANE BACKGROUNDS (SAVAGE)

It's time to switch our focus back to arcane backgrounds so we can continue fleshing out their behavior.

MUTUAL EXCLUSION

The Savage Worlds rules utilize the same basic mechanics for all types of arcane backgrounds, with the only difference being the trappings associated with the powers. As such, there is no practical benefit for a character to choose more than one arcane background, so we make the assumption that characters cannot choose multiple arcane backgrounds. This will make things much simpler for us to manage in the data files.

Imposing mutual exclusion on the various arcane backgrounds requires a little bit of work. Since arcane backgrounds can only be added via the various edges, we need to control the exclusion on the edges. One way of solving this would be to use a ContainerReq tag expression on each arcane background edge. This approach is optimal when a pick must be effectively ignored when the user makes an independent change to the character, but it is otherwise a more complex solution. In our case, all the logic for selecting arcane backgrounds is managed through the edges, so we simply need to ensure that only a single edge is ever added to the character at one time. If the user adds an arcane background and wants to change it to something else, he must delete the first one and can then add the new one back in.

Controlling this is achieved through the Candidate tag expression on the table portal, which governs the list of things that the user can choose from. The table of edges doesn't currently have a Candidate tag expression, so we'll need to add one. Candidate tag expressions can either be designed to include a subset of things that satisfy certain criteria or to exclude a subset that meet the criteria. Since we want to include all arcane backgrounds by default, our Candidate test needs to be designed to exclude things only when certain conditions exist.

We now need to figure out what tags we'll be filtering on. We need to limit our exclusion logic to arcane background edges. Looking at the various arcane background edges, there is nothing that identifies them clearly as arcane backgrounds. We need some way to do this, so we might as well assign the appropriate "Arcane" tag to each one. We can then focus our exclusion test only on edges that possess an "Arcane.?" tag. Or, by reversing the logic, we can exempt all edges lacking the tag from our exclusion test. Half of our test is now figured out. Just be sure to open the file "thing_edges.dat" and assign the appropriate tag to each arcane background edge.

If the user has already selected an arcane background for the character, we want to hide all others from selection. So we now need to figure out how to determine whether the user has selected an arcane background already. Fortunately, this is easy. Any time an

arcane background is added, the "Arcane.?" tag is automatically forwarded to the character. This means we can check the character for one of the tags and instantly determine whether an arcane background has already been added.

Both halves of the problem have been solved. All that's left is for us to piece the tag expression together properly. The first half is to exempt from our tests any edges that lack an arcane background tag, so that translates to "!Arcane.?". This limits our focus in the rest of the tag expression to edges that are arcane backgrounds. The second half is to allow any arcane background if none has yet been added, and that translates to "!Arcane.?". But wait a minute. That's the same test, isn't it? Yes and no. It's the same test, but this time it needs to be applied to the character directly. By default, the Candidate tag expression is applying the tests to the thing that will potentially be shown to the user for selection. For the second half of our tag expression, we need to explicitly perform the test on the character instead of the thing. This is accomplished by prefixing the tag template with "hero#". Putting it all together yields the final Candidate tag expression below.

```
<candidate>!Arcane.? | !hero#Arcane.?</candidate>
```

Re-load the data files and go to the "Edges" tab. Click the option to add a new edge, and you should see all five arcane backgrounds listed properly. Add one to the character and the remaining arcane backgrounds all disappear. Exit the chooser form and delete the arcane background you just added. Now go back in and they are all visible again for selection. Our mutual exclusion logic is working.

Hiding Arcane Skills

There is one glaring problem with the solution we've put into place regarding arcane skills. All of the various arcane skills are always visible for user selection, regardless of whether the character possesses the corresponding arcane background. It would be vastly better to only show the proper arcane skill for selection on the "Skills" tab if the character actually has the proper arcane background. One way of doing this would be to revise the Candidate tag expression on the table of skills to only let the user select the skill if the background is present. The drawback of this approach, though, is that a user can add an arcane background, add the skill, then delete the background, and the skill will continue to exist without any warning to the user.

We're going to employ an even better solution by leveraging a ContainerReq tag expression. The ContainerReq test is applied to the prospective container when things are being selected by the user. It is also applied to the actual container when picks have been added to the character. If the ContainerReq test fails when showing a thing for user selection, the thing is treated as if it doesn't exist, so it is physically omitted from the list of things presented to the user. If a thing is added and then its ContainerReq test fails due to other changes to the character, the pick is disabled by HL, the pick can be easily highlighted as invalid, and a suitable validation error is automatically reported. The pick still remains visible to the user, but that's basically just so that the user can respond to the error by deleting the pick.

When a ContainerReq test is defined for a thing, Hero Lab must be told when to perform the test within the overall evaluation cycle. The ContainerReq test must be the very first task performed for the thing, so it must be scheduled rather early in the overall cycle. However, it's quite possible that other tasks need to be processed first within the evaluation cycle. In our case, that's important, since our ContainerReq test needs to check against the presence of the "Arcane" tag on the character, but a script is needed to forward the tag from the arcane background up to the character. Consequently, we need to forward the tag from the arcane background before the ContainerReq test is performed.

If we want to find out the latest point in the evaluation cycle that we can schedule our ContainerReq test, we can add a ContainerReq test that is scheduled to occur at an extremely late timing (e.g. Render/10000). By re-compiling, HL will report an error, telling us when the earliest task on the thing is scheduled. Doing that yields an error that states we need to schedule our ContainerReq test at a timing of Initialize/3000 or earlier. We'll use Initialize/2000, which is specified via the "phase" and "priority" attributes. This results in a revised "Spellcasting" thing that looks like the following.

```
<thing
  id="skSpellcst"
  name="Spellcasting"
  compset="Skill"
  isunique="yes"
  description="Description goes here">
  <fieldval field="trtAbbrev" value="Spl"/>
  <tag group="Arcane" tag="Magic"/>
  <containerreq phase="Initialize"
priority="2000">Arcane.Magic</containerreq>
  <link linkage="attribute" thing="attrSma"/>
  </thing>
```

NOTE! The above definition might seem a bit odd at first, since it defines the tag "Arcane.Magic" and has a ContainerReq test on the same tag. The important distinction is that the tag assignment assigns the tag to the thing, while the ContainerReq test checks for the tag on the container of the thing/pick. Since skills are always added directly to the character, the container is the character. This means that the tag is added to the thing but the test is performed on the character.

We have a problem, though. If we re-load the data files and check the table of skills, the "Spellcasting" skill will never appear, even if we properly select the "Arcane Background: Magic" edge. The problem is our timing, since the "Arcane" tag is still being forwarded by the "Arcane" component at the timing we previously assigned (Initialize/3000). We need to revise the timing of this script so that it occurs before the ContainerReq test. We'll use Initialize/1000, which yields a revised script of the following.

```
<eval index="1" phase="Initialize" priority="1000"><![CDATA[
  perform forward[Arcane.?]
  ]]></eval>
```

Everything should now work. Until the arcane background edge is selected, the corresponding skill never appears. Once the edge is added, the skill appears and can be added. If the user deletes the edge, the skill turns red and a validation error is reported until we delete it (or re-add the edge).

Arcane Power Mechanics

The mechanics for arcane backgrounds are already in place, but we still need to add the mechanics for arcane powers. This entails adding a new component and a new component set. Arcane powers have an assortment of characteristics that need to be tracked. We'll manage them all via fields for simplicity. All of these fields can be customized in some way beyond just tracking a number, except for

the minimum rank required, so they need to be treated as text-based fields. Putting it all together yields the following component and component set, which can both be added to the file "miscellaneous.str".

```
<component
  id="Power"
  name="Arcane Power"
  autocompset="no">
   <field
    id="powMinRank"
    name="Minimum Rank"
    type="static">
    </field>
   <field
    id="powPoints"
    name="Power Point Cost"
    type="static"      maxlength="25">
    </field>
   <field
    id="powRange"
    name="Range"
    type="static"
    maxlength="25">
    </field>
   <field
    id="powLength"
    name="Duration"
    type="static"
    maxlength="25">
    </field>
   <field
    id="powMaint"
    name="Maintenance"
    type="static"
    maxlength="25">
    </field>
   <field
    id="powTraps"
    name="Trappings"
    type="user"
    maxlength="100">
    </field>
  </component>

<compset    id="Power">
  <compref component="Power"/>
  </compset>
```

There is one thing missing from the above component, though. Each component needs to properly consume one instance from the resource that tracks the number of available powers. This ensures that the number of powers left decreases as new powers are added to the character, and it can be accomplished with a simple component-based Eval script like the one shown below.

```
<eval index="1" phase="Setup" priority="5000"><![CDATA[
  #resspent[resPowers] += 1
  ]]></eval>
```

RE-USE OF MINIMUM RANK

Powers have a minimum required rank that behaves exactly the same as for edges. That leaves us with two options. First, we could copy the logic over and adapt it for powers. Second, we could look for a way to carve out the minimum rank logic into a separate component that could be easily re-used. Since the minimum rank logic is identical for edges and powers, and since proper handling of the minimum rank entails testing the pre-requisites, it makes much more sense to carve out the logic for re-use.

Extracting the logic entails creating a new component that we'll call "MinRank". This new component needs to possess a single field for tracking the minimum rank and the pre-requisite logic. So we move the "edgMinRank" field to the new component and change its name to "rnkMinRank". Then we move the pre-requisite test to the new component. This results in a component that looks like the following.

```
<component
  id="MinRank"
  name="Minimum Rank"
  autocompset="no">
  <field
    id="rnkMinRank"
    name="Minimum Rank"
    type="static">
    </field>

 <prereq iserror="yes" message="Veteran Rank required.">
   <valid><![CDATA[
   ~get the minimum rank required for the thing to be valid
   var rank as number
   rank = altthing.field[rnkMinRank].value

   ~if the minimum rank is satisfied, we're good to go
   if (herofield[acRank].value >= rank) then
     @valid = 1
     done
     endif

   ~mark the panel as invalid
   altthing.linkvalid = 0

   ~synthesize an appropriate validation error message
   if (rank = 1) then
     @message = "Seasoned"
   elseif (rank = 2) then
     @message = "Veteran"
   elseif (rank = 3) then
     @message = "Heroic"
   elseif (rank = 4) then
     @message = "Legendary"
     endif
   @message &= " rank required."
   ]]></valid>
   </prereq>
  </component>
```

Unfortunately, we can't put the new component in the file "traits.str". We need to use this component in the component set for edges (in "traits.str") and the component set for powers (in "miscellaneous.str"). Since the compiler processes files with the same file extension (e.g. ".str") in whatever order that Windows gives them to HL, and since that order is not consistent across versions of Windows, we can't safely put the component in one file or the other. Instead, it needs to be placed in a file that is guaranteed to be compiled prior to both files. Such a file already exists as the file "components.core", which already contains a few components that are shared by different component sets. So we add our new component to the file "components.core", and everything should work smoothly.

Once the component is defined, we need to add it to the two component sets. This is easily achieved by adding the line below to both component sets.

```
<compref component="MinRank"/>
```

The last thing we need to do is deal with the fact that we've renamed the field for use within a different component. That means that all of our existing edges must have the field properly renamed. Open the file "thing_edges.dat" and perform a search-and-replace operation that changes all instances of "edgMinRank" to "rnkMinRank". After that, the logic has been factored out and is now being cleanly re-used in multiple places.

Adding Arcane Powers

Now that all the mechanics are in place, we can add the actual arcane powers. Since we're putting all arcane content in the file "thing_arcane.dat", that's where we'll add the powers. No arcane powers have special behaviors that we need to handle, so we'll present one as an example below. The rest can be defined following this example, or you can find them all in the completed Savage Worlds data files.

```
<thing
  id="powArmor"
  name="Armor"
  compset="Power"
  isunique="yes"
  description="Description goes here">
  <fieldval field="rnkMinRank" value="0"/>
  <fieldval field="powPoints" value="2"/>
  <fieldval field="powRange" value="Touch"/>
  <fieldval field="powLength" value="3"/>
  <fieldval field="powMaint" value="1/round"/>
  <fieldval field="powTraps" value="A mystical glow..."/>
  </thing>
```

Add an "Arcane" Tab (Savage)

With the basics of arcane powers now in place, we need to add a suitable tab where they can be managed by the user. We'll also need this new tab to adequately test the additional handling we must still implement for arcane powers.

Something Simple to Start With

We'll use the same technique that we used previously for obtaining a simple new tab. We'll clone the "tab_skills.dat" file and convert it over for handling arcane background details. Since the primary thing we'll be showing in this tab is a table of arcane powers, we'll convert the table over for that purpose. We'll also make use of the "SimpleItem" template within the table for expedience. Arcane powers will require that we provide our own custom template to show the various fields, but this will serve as a useful placeholder for the moment.

When the conversion is done, we'll have a single portal, layout, and panel. All of them will be tailored for their new purpose, ultimately looking something like the following.

```
<portal
  id="apPowers"
  style="tblNormal">
  <table_dynamic
    component="Power"
    showtemplate="SimpleItem"
    choosetemplate="SimpleItem"
   addpick="resPowers">
    <titlebar><![CDATA[
      @text = "Add an Arcane Power - " &
     ]]></titlebar>
    <headertitle><![CDATA[
      @text = "Arcane Powers - " &
hero.child[resPowers].field[resSummary].text
     ]]></headertitle>
    <additem><![CDATA[
      ~get the color-highlighted "add" text
      @text = field[resAddItem].text
     ]]></additem>
    </table_dynamic>
  </portal>

 <layout
   id="arcane">
   <portalref portal="apPowers" taborder="10"/>
   <position><![CDATA[
    ~freeze our table in advancement mode to disable adding
new choices
    ~Note: All freezing must be done *before* any positioning is
performed.
    if (state.iscreate = 0) then
     portal[apPowers].freeze = 1
     endif

   ~position and size the table to span the full layout; it will only
use the
    ~vertical space that it actually needs
    perform portal[apPowers].autoplace
   ]]></position>
  </layout>

 <panel
   id="arcane"
   name="Arcane"
   marginhorz="5"
   marginvert="5"
   order="140">
   <layoutref layout="arcane"/>
   <position><![CDATA[
    ]]></position>
   </panel>
```

Panel Visibility

We need to address when the "Arcane" tab appears to the user. Since the panel only applies for a character that has selected an arcane background, the panel should only be shown when a character possesses such a background. The visibility of the panel can be controlled via the Live tag expression, and that tag expression can key on whether the character possesses any "Arcane" tag. This can be implemented by adding the following line to the panel definition.

```
<live>Arcane.?</live>
```

Panel Linkage

Arcane powers are exclusively selected via the "Arcane" tab. As such, any errors that may arise with arcane powers should highlight the "Arcane" tab in red so the user knows where to correct them. We can automatically associate every arcane power with the "Arcane" tab by specifying the "panellink" attribute within the "Arcane" component. Set the "panellink" attribute to "arcane" to establish the linkage for all powers.

Refining Arcane Backgrounds and Powers (Savage)

There are a variety of special restrictions and handling that need to be performed for arcane backgrounds and powers. Now that we've got the "Arcane" tab in place and working, we can add and test the remaining logic.

Arcane Background Drawbacks

Certain arcane backgrounds possess drawbacks that occur when the character rolls poorly when activating an arcane power (e.g. backlash, brainburn, etc.). These drawbacks are fundamental to the character, so they need to be shown on the "Special" tab. This entails we take a number of steps.

The first thing we need to do is define a new component and component set for drawbacks. The component set will integrate both the new "Drawback" component and the "SpecialTab" component so that all derived things appear properly on the "Special" tab. Drawbacks don't have any fields, but they need to have a unique "SpecialTab" tag to control their positioning on the "Special" tab, so we define the tag within the file "tags.1st" and then the component can assign that appropriate tag. The net result is something similar to the following.

```
<component
  id="Drawback"
 name="Drawback for Arcane Background"
  autocompset="no">
  <tag group="SpecialTab" tag="Drawback"/>
  </component>

<compset
  id="Drawback">
  <compref component="Drawback"/>
  <compref component="SpecialTab"/>
  </compset>
```

Once the component and component set are in place, we need to define each of the drawbacks. They are all extremely simple, requiring nothing special at all. We define them in the file "thing_arcane.dat", along with all the other arcane mechanics. We should also define them close to each of the arcane backgrounds to which they correspond. The following is an example of a drawback.

```
<thing
  id="drwMagic"
  name="Backlash"
  compset="Drawback"
 isunique="yes"
  description="Description goes here">
  </thing>
```

The final step we need to do is to automatically add the proper drawback whenever the corresponding arcane background is added. This is accomplished by specifying a "bootstrap" element within the arcane background. The "bootstrap" element automatically adds the designated thing whenever HL creates the thing it is assigned for. So the revised "Magic" arcane background would look something like the following.

```
<thing
  id="arcMagic"
  name="Magic"
  compset="Arcane"
  isunique="yes"
  description="Description goes here">
  <fieldval field="arcPowers" value="3"/>
  <fieldval field="arcPoints" value="10"/>
  <bootstrap thing="drwMagic"/>
  </thing>
```

Non-Available Powers

Various arcane powers are not suitable for use with certain arcane backgrounds. Since the rules outline these restrictions as suggestions, it would be inappropriate for the data files to prohibit arcane powers for arcane backgrounds. Instead, we'll use the Kit's pre-requisites mechanism to flag arcane powers as improper, which will still allow them to be taken and simply present a validation error.

The simplest and most obvious way to handle this is to write separate pre-requisites for every arcane power, specifically testing against the precluded arcane backgrounds. But that requires that we define a lot of very similar pre-requisites. It also means that any supplements that introduce new arcane powers will similarly have to define all the separate pre-requisites. That's a lot of work that we can avoid by writing a single pre-requisite test on the component that is shared by all arcane powers.

Doing this requires that we devise a way to conveniently identify the various combinations of arcane powers and backgrounds that are either valid or invalid. Since most powers are available to each background, the list of exceptions will be much easier to maintain. Then the question becomes whether it's easier to track the precluded powers for each background or the precluded backgrounds for each power. The rules present the list of precluded powers for each background, so this would make sense, as it follows the presentation of the rulebook. However, stop and think for a moment about how this would be implemented. It would require that every arcane power be sanity checked against information that is part of the arcane background, and that would require a good amount of extra work to manage in the data files. In contrast, having each arcane power identify the arcane backgrounds to which it doesn't apply would be easy to implement, since the character already has a suitable arcane background tag assigned that can be checked against. So we'll go with this latter approach.

With our design in mind, we need a way to designate an arcane power as being invalid for a particular arcane background. Tags are the obvious solution, and we already have an "Arcane" tag group that we could use. However, the "Arcane" tag group is already used to identify what something is, so it would be confusing to use it at other times to indicate what something is not. The easiest way to solve this is to create a new tag group that indicates an arcane background that a power is not allowed to be. We'll call the new tag group "ArcaneDeny" and define the exact same set of tags as for the "Arcane" tag group, resulting in something like below.

```
<group
  id="ArcaneDeny"
  dynamic="yes">
  <value id="Magic"/>
  <value id="Miracles"/>
  <value id="Psionics"/>
  <value id="SuperPower"/>
  <value id="WeirdSci"/>
  </group>
```

Now we can assign the appropriate "ArcaneDeny" tags to all of our powers. For example, the rulebook states that the "Magic" arcane background does not usually have access to "Healing" or "Greater

Healing". Consequently, we'll need to assign the "ArcaneDeny.Magic" tag to both of those powers. The "Healing" power is shown below as an example.

```
<thing
  id="powHealing"
  name="Healing"
  compset="Power"
  isunique="yes"
  description="Description goes here">
  <fieldval field="rnkMinRank" value="0"/>
  <fieldval field="powPoints" value="3"/>
  <fieldval field="powRange" value="Touch"/>
  <fieldval field="powLength" value="Instant"/>
  <fieldval field="powMaint" value=""/>
  <fieldval field="powTraps" value="Laying on hands, touching
the victim with a holy symbol, prayer"/>
  <tag group="ArcaneDeny" tag="Magic"/>
  <tag group="ArcaneDeny" tag="SuperPower"/>
  <tag group="ArcaneDeny" tag="WeirdSci"/>
  </thing>
```

At this point, every power should properly identify the arcane background(s) for which it is not valid. The final thing we need to do is define the appropriate pre-requisite test to verify whether a given power is valid. Since each power identifies any backgrounds that it is invalid for, the pre-requisite must determine if the list of forbidden backgrounds matches the active background for the character. If it does, then the power is considered invalid. This can be accomplished via use of the "tagmatch" target reference in the Validate script, which can be used to check whether the "Arcane" tag on the character (i.e. the arcane background) matches any corresponding "ArcaneDeny" tags on the power. The resulting pre-requisite should look something like the following.

```
 <prereq message="Not suitable for character's arcane
background.">
   <valid><![CDATA[
    ~look for an "Arcane" tag in the initial context (character); if
found, look for
    ~a corresponding "ArcaneDeny" tag in the power; if a
matching tag is found, it
    ~means the power is invalid, so a result of zero indicates
we're valid
    var result as number
    result = altthing.tagmatch[Arcane,ArcaneDeny,initial]
    if (result = 0) then
      @valid = 1
      done
      endif
    ~mark the linked panel as invalid
    altthing.linkvalid = 0
    ]]></valid>
   </prereq>
```

PRECLUDED EDGES

The rules for Weird Scientists explicitly preclude the ability to select the "Soul Drain" edge. This means that the two require a suitable pre-requisite to deny the other. The pre-requisite for each needs to simply verify that the other edge has not already been selected, which can be handled via the "pickreq" element. The pre-requisite for the "Weird Science" arcane background edge will look something like the following, with the one for the "Soul Drain" edge looking very similar. [Remember: This is applied to the edge and not the arcane background.]

```
<pickreq ispreclude="yes" thing="edgSoulDrn"/>
```

SUPER POWERS

The "Super Powers" arcane background doesn't have a corresponding arcane skill. Instead, every single arcane power has its own separate arcane skill when used as a super power. This requires that we define a suitable arcane skill for every arcane power. We also need to make sure these arcane skills are only visible when the corresponding power is possessed by the character and the arcane background employed is "Super Powers". We can use the same technique employed for normal arcane skills (i.e. a ContainerReq test), although we now need to test for a specific arcane power. There is no indication on the character of which arcane powers are possessed, so we'll need to add that.

The easiest way to identify which arcane powers have been chosen on the character involves the use of identity tags. The "Power" component can define identity tags for each individual power. Then a component-based Eval script can forward the identity tags of added powers up to the character. Once the tags are on the character, they can be readily tested by the ContainerReq test. The code below shows what must be added to the "Power" component. Note that we've scheduled the script to occur at the same timing as the script that forwards the "Arcane" tag for arcane backgrounds (Initialize/1000). Since we'll be depending on these tags within ContainerReq tests using the same logic, it's easiest to keep our timing consistent within the data files.

```
<identity group="Power"/>

<eval index="2" phase="Initialize" priority="1000"><![CDATA[
  perform forward[Power.?]
  ]]></eval>
```

We're now ready to define the arcane skills for each power. Or are we? It seems we have a problem, because arcane skills for super powers do not have an associated linked attribute. However, we've defined skills such that they require a linked attribute. One obvious solution would be to eliminate the requirement that skills always have a linked attribute. However, that would make it easy for anyone adding a new skill to inadvertently forget to add the linkage and we'd have to handle missing linkages. A less obvious (but better) solution would be to define a special attribute that was always hidden from the user. This provides us with a linked attribute for which we can control the value, ensuring that skills linked to it are always treated appropriately with regards to advancement costs. This is a little bit more work, but it provides greater safety, so we'll use this approach.

Adding the new attribute entails a number of minor tasks. First of all, the attribute needs to be hidden, so we'll assign it the pre-defined "Hide.Attribute" tag for this purpose. Next, we need to setup the value of the attribute appropriately so that the skills properly determine whether they are less than the linked attribute. Although the rulebook isn't clear on this point, the Savage Worlds designers have confirmed that skills for super powers are always considered greater than or equal to the linked attribute, which means that we can safely leave the attribute at its default value. Lastly, the attribute itself must be defined in the file "thing_attributes.dat", as shown below.

```
<thing
  id="attrSup"
  name="Super Power"
  compset="Attribute"
  isunique="yes"
  description="Used for arcane skills based on powers for
characters with the Super Powers arcane background.">
  <tag group="Hide" tag="Attribute"/>
  </thing>
```

Once we have the new attribute in place, we can finally define our new arcane skills tied to the arcane powers. All of the skills will be handled the same way, so only one is presented below as an example. The skill needs to perform a ContainerReq test that verifies both the arcane background and the proper arcane power.

```
<thing
  id="skpArmor"
  name="Armor (Power)"
  compset="Skill"
  isunique="yes"
  description="Description goes here">
  <fieldval field="trtAbbrev" value="Armr"/>
  <tag group="Arcane" tag="SuperPower"/>
  <containerreq phase="Initialize" priority="2000"><![CDATA[
    Arcane.SuperPower & Power.powArmor
    ]]></containerreq>
  <link linkage="attribute" thing="attrSup"/>
  </thing>
```

Synthesizing Descriptions for Display (Savage)

The description text that is shown to the user often includes more than the actual description text entered for each thing. If the thing has fields, the description text viewed by users will typically be improved by integrating the values of some or all of those fields. This synthesized description text is used from all sorts of places within HL. Fortunately, the Skeleton data files integrate a centralized mechanism for synthesizing description text for virtually all types of things. This makes it relatively easy to extend the mechanism for use with your own custom components.

Using Procedures

At the hub of the description logic is a re-usable procedure with the id "Descript". This procedure is found in the file "procedures.dat" and orchestrates the synthesis of appropriate description text for just about any type of thing. Whenever you want to display the description text of a thing, your script can call this procedure to do the work. Then you can use the synthesized result for display to the user. The key advantage of using a centralized mechanism like this is that you only have to write the logic for synthesizing the description once, after which you can re-use it from anywhere. In addition, since the one mechanism intelligently supports all different things, you don't have to remember to invoke the correct logic from the correct places - just use the same procedure from everywhere.

The "Descript" procedure utilizes a standard sequence in which it pieces together a description for output. This is good for users, because all the information appears in a consistent fashion for each type of thing. The procedure keys on facets of the thing to determine the nature of what it's processing and selectively includes/excludes different types of information. For example, if the thing is gear, appropriate gear details are included. Similarly, failed pre-requisite details are only output if a thing is being processed (instead of a pick).

In the middle of the procedure, the nature of the thing is keyed upon to call an appropriate helper procedure. This allows the specifics of each particular type of thing to be encapsulated into a separate procedure. It accomplishes this by keying on an appropriate "component" tag and calling the associated procedure. Suitable helper procedures for a handful of different components are provided within the Skeleton files for use by the "Descript" procedure. As part of all this, an "InfoWeapon" procedure is defined that handles all generic details for all types of weapons. Both the "InfoMelee" and "InfoRange" procedures call "InfoWeapon" to output their common aspects. The net result is a well-organized structure that is easily maintained and easily extended.

You'll also see a "MouseInfo" procedure within the file. This procedure is designed for use by MouseInfo scripts to synthesize the output for display. It is designed for a slightly different output display, but it calls the "Descript" procedure to do the vast majority of output synthesis.

Integrating the "Descript" Procedure

Anywhere that you want to display the description text for a thing or pick, you can call the "Descript" procedure and use its results. The procedure is designed for use from any "info" script context as opposed to only Description scripts, so it is unable to use any special symbols. Instead, the synthesized results are placed into a "descript" string variable. If the caller wants to access the results, it must also define the same variable prior to calling the procedure. So the typical use of the "Descript" procedure amounts to the following script code.

```
var descript as string
call Descript
@text = descript
```

The "Descript" procedure has one optional parameter that you can specify. Normally, the list of failed pre-requisites is only included when synthesizing the description for a thing. However, there may be times where you want to include failed pre-requisites for a pick. In this situation, you can force the procedure to include the extra information via the "isprereq" numeric variable. Since variables are inherited by called procedures, you can define the "isprereq" variable, set it to a non-zero value, and then call the script. An example of this usage is shown below.

```
var descript as string
var isprereq as number
isprereq = 1
call Descript
@text = descript
```

NOTE! Since numeric variables always default to zero, omitting the variable is the same as defining it to zero. So we normally just omit it, unless there is an important reason to make the default behavior explicitly clear.

Integrating the "MouseInfo" Procedure

The "MouseInfo" procedure is used just like the "Descript" procedure, except that it is designed for use from within MouseInfo scripts. The synthesized results from this procedure are placed in a string variable named "mouseinfo", but it otherwise works no differently. The "isprereq" variable is also supported. Consequently,

the typical usage of the "MouseInfo" procedure looks like the script code shown below.

```
var mouseinfo as string
call MouseInfo
@text = mouseinfo
```

Putting "Descript" and "MouseInfo" to Use

Now that we have our "Arcane" tab in place, we're going to want to show more than just the name and description text for the various arcane powers. We're also going to want to see the power point cost, range, duration, etc. All that information is specified via fields, which won't get displayed by default. So we need to utilize our own Description script for the table portal and that means we'll put the "Descript" and "MouseInfo" procedures to use.

But wait. HL is already using the "Descript" and "MouseInfo" procedures. By default, the Kit automatically hooks up these two procedures as the default behaviors used whenever a Description script or MouseInfo script is required. If we don't specify anything specific for one of these scripts, the "Descript" or "MouseInfo" procedure, as appropriate, will be invoked. So we don't have to do anything special to integrate our special handling logic.

Extending the "Descript" Procedure

In keeping with the conventions used in the Skeleton files, we're going to extend the "Descript" procedure to include detailed output for arcane powers. To do this, we'll need to add a new procedure that synthesizes the custom details appropriately. We also need to call the procedure properly from within the "Descript" procedure. We'll name our new procedure "InfoPower" to match the component and start by splicing it into the "Descript" procedure. Looking at the "Descript" procedure, you'll see a section of script code that checks for various component tags and calls a different procedure for each. That's where we want to add our new code. At the end of the "if/then/else" block, we'll add a test for the "Power" component and call our new procedure. The resulting code block should look like the following.

```
iteminfo = ""
if (tagis[component.WeapRange] <> 0) then
  call InfoRange
elseif (tagis[component.WeapMelee] <> 0) then
  call InfoMelee
elseif (tagis[component.Defense] <> 0) then
  call InfoDef
elseif (tagis[component.Power] <> 0) then
  call InfoPower
endif
```

Now we need to add our new procedure. We'll place it immediately beneath the "Descript" procedure to keep all the info-related mechanisms together. Our new procedure needs to output the various fields that are unique to arcane powers on separate lines. We'll use a format similar to the one used in the rulebook so that everything looks familiar to the user. That means blending any maintenance cost into the duration. Since the trappings are designed to be edited by the user once the power is added, we need to gracefully handle a situation where the user has left the trappings empty. The net result is a procedure that looks like the one below.

```
<procedure id="InfoPower" context="info"><![CDATA[
  ~declare variables that are used to communicate with our
  var iteminfo as string
  iteminfo = ""

 ~report the power point cost
 iteminfo &= "Power Points: " & field[powPoints].text & "{br}"

 ~report the range
  iteminfo &= "Range: " & field[powRange].text & "{br}"

 ~report the duration and any maintenance cost (omitting if
there is none)
  iteminfo &= "Duration: " & field[powLength].text
  if (field[powMaint].isempty = 0) then
    iteminfo &= " (" & field[powMaint].text & ")"
    endif
  iteminfo &= "{br}"

 ~report the trappings for the power
  iteminfo &= "Trappings: "
  if (field[powTraps].isempty <> 0) then
    iteminfo &= "n/a"
  else
    iteminfo &= field[powTraps].text
    endif
  iteminfo &= "{br}"
  ]]></procedure>
```

Once our new procedure is in place, re-load the data files and click on the table to add a new arcane power. In the description window on the right, you'll see all of the fields properly listed for each arcane power.

Refining the "Arcane" Tab (Savage)

The "Arcane" tab is currently very minimalist in nature. None of the special fields of powers are shown for selected powers. We also don't show any summary information about the arcane background on the tab, and we really should be doing that. It's time for us to open the file "tab_arcane.dat" and start refining it.

Selecting Arcane Powers

The selection of arcane powers is working properly, but the display has one aspect that we should address. Arcane powers typically have lots of description text associated with them. Consequently, the relatively narrow space that is provided by default when selecting powers should be widened. This can be easily achieved by adding a new attribute to the table portal, within the "table_dynamic" element. The new attribute is "descwidth" and controls the pixel width to be used when allowing the user to select items for the table. We'll choose a width of "350" to make the paragraphs of description text easier to read.

```
 <portal
   id="apPowers"
  style="tblNormal">
  <table_dynamic
    component="Power"
    showtemplate="apPower"
    choosetemplate="SimpleItem"
    addpick="resPowers"
    descwidth="350">
    <titlebar><![CDATA[
      @text = "Add an Arcane Power - " &
hero.child[resPowers].field[resSummary].text
      ]]></titlebar>
    <description/>
    <headertitle><![CDATA[
      @text = "Arcane Powers - " &
hero.child[resPowers].field[resSummary].text
```

```
    ]]></headertitle>
   <additem><![CDATA[
    ~get the color-highlighted "add" text
    @text = field[resAddItem].text
    ]]></additem>
   </table_dynamic>
  </portal>
```

Showing Arcane Powers

Once an arcane power is added to the character, all the user sees is its name. It would be quite helpful to also see the power point cost, range, and duration for the power. In addition, we need to allow the user to specify the actual trappings for the power. None of this can be accomplished with the "SimpleItem" template that we're currently using, so we need to switch that out for a custom template. We need to start with a template that is extremely simple and that we can readily adapt. The "SimpleItem" template is an excellent candidate, but the template for edges is even simpler, so we copy that template from the file "tab_edges.dat". After the copy, all we need to change is the unique id, name, and component set. This gives us something simple to start with like the example below.

```
<template
  id="apPower"
  name="Power Pick"
  compset="Power"
  marginhorz="3"
  marginvert="2">

 <portal
   id="name"
   style="lblNormal"
   showinvalid="yes">
   <label
    field="name">
    </label>
  </portal>

 <portal
   id="info"
   style="actInfo">
   <action
    action="info">
    </action>
  <mouseinfo mousepos="middle+above"/>
  </portal>

 <portal
   id="delete"
   style="actDelete"
   tiptext="Click to delete this item">
   <action
    action="delete">
    </action>
  </portal>

 <position><![CDATA[
   ~set up our height based on our tallest portal
   height = portal[info].height

  ~if this is a "sizing" calculation, we're done
  if (issizing <> 0) then
    done
    endif

  ~position our tallest portal at the top
  portal[info].top = 0

  ~position the other portals vertically
  perform portal[name].centervert
  perform portal[delete].centervert
  ~position the delete portal on the far right
  perform portal[delete].alignedge[right,0]

  ~position the info portal to the left of the delete button
  perform portal[info].alignrel[rtol,delete,-8]

  ~position the name on the left and use all availble space
  portal[name].left = 0
  portal[name].width =
minimum(portal[name].width,portal[info].left - portal[name].left -
10)
   ]]></position>
  </template>
```

When we add the various fields for arcane powers to what is shown, we can either use separate portals for each field or we can combine them all into a single portal. In the case of arcane powers, we don't need to do anything fancy when presenting the information. Consequently, we simply define a single label portal that is driven by a script. The script will combine the different fields into a useful string of text for display. An example of what such a portal might look like is shown below.

```
 <portal
   id="details"
   style="lblSmall">
   <label
    multiline="yes">
    <labeltext><![CDATA[
     @text = "{/b}Points:{b} " & field[powPoints].text
     @text &= "{br}{/b}Range:{b} " & field[powRange].text
     @text &= "{br}{/b}Duration:{b} " & field[powLength].text
     if (field[powMaint].isempty = 0) then
      @text &= " (" & field[powMaint].text & ")"
      endif
    ]]></labeltext>
   </label>
  </portal>
```

Note that we could have strung all of the fields together on a single line, but then we wouldn't always have room for them all. So we instead string them together with a newline ("{br}") between each, which makes sure that everything is visible. Since this is a multi- line label, we need to make sure to designate it as such so that HL handles it properly for auto-sizing purposes.

Before we revise the template Position script to re-position the portals properly, let's also address the ability for users to specify the trappings for each power. The description of the trappings for a power may be somewhat lengthy, so our first inclination is to use a wide edit portal for the purpose, or perhaps even a multi-line edit portal. However, trappings are not something that the user will likely want to see all the time, so dedicating a large area of screen real estate to trappings is probably not a good idea. A better approach is to do something similar to the way details are handled for Journal entries: an "edit" button allows the user to edit the contents via a dedicated form.

So our plan is to clone the portal used for journal entries and adapt it to our needs. Open the file "tab_journal.dat" and locate the portal with a unique id of "notes". This is the portal used for editing the notes of journal entries. Copy the portal and paste it into the arcane power template that we're developing. We can now give it a more suitable unique id ("trappings"), change the tip text, and associate it with the proper field for the trappings ("powTraps"). This portal will provide a button that the user can click on, which will bring up

a separate form where the trappings of the power can be edited. The new portal should look similar to the example below.

```
<portal
  id="trappings"
  style="actNotesSm"
  tiptext="Click here to edit the trappings for the arcane
power.">
  <action
    action="notes"
    field="powTraps">
    </action>
  </portal>
```

We can now properly position all of our portals. The tallest portal will be the details, and it will always have a height of a full three lines of text, so that will dictate our template height. One important detail, though, is that the initial "sizing" handling of templates within tables performs its task generically (i.e. without a specific pick to process). As a result, dynamic calculations like "textheight" on a script-based label will always retrieve the height for an empty string. This means that we cannot use "textheight" to determine the height of the portal during "sizing" and must rely on "fontheight" instead. Continuing on, we'll center the name vertically, then we'll stack the various buttons on the right, with the "delete" and "info" portals on top and the "trappings" portal beneath. Putting it all together yields a Position script similar to the one below.

```
~set up our height based on our tallest portal
 ~Note: Since the portal is a script-based label, it will not have
any contents
 ~when the height of the template is initially determined for
general sizing.
 ~Consequently, we can't use "textheight" here and must use
"fontheight" instead.
 height = portal[details].fontheight * 3

~if this is a "sizing" calculation, we're done
 if (issizing <> 0) then
   done
   endif

~position our tallest portal at the top
 portal[details].top = 0

~center the name vertically
 perform portal[name].centervert

~center the info and delete portals within the upper half of the
template
 var half as number
 half = height / 2
 portal[info].top = (half - portal[info].height) / 2
 portal[delete].top = (half - portal[delete].height) / 2

~center the trappings portal within the lower half of the template
 ~Note: The info portal is big, so shift it down a little extra for
better spacing.
 portal[trappings].top = half + (half - portal[trappings].height) / 2
+ 1

~position the delete portal on the far right
 perform portal[delete].alignedge[right,0]

~position the info portal to the left of the delete button
 perform portal[info].alignrel[rtol,delete,-6]

~center the trappings portal between the info and delete buttons
 var span as number
 span = portal[delete].right - portal[info].left
 portal[trappings].left = portal[info].left + (span -
portal[trappings].width) / 2
~position the name on the left and use a suitable amount of
space
 portal[name].left = 0  portal[name].width = 200

~position the details next to the name and use the remaining
space
 perform portal[details].alignrel[ltor,name,10]
 portal[details].width = portal[info].left - portal[details].left - 10
```

Refining Arcane Powers

Now that everything is working, we need to refine the behaviors. The first thing we need to address is that centered text just doesn't look very good for either the power name or the details. Open the file "styles_ui.aug", which is where all the styles are defined that are used within the user-interface. Scroll down to where the label styles are defined and scan through the pre-defined options that are available. For the power name, we need something in a "Large" font, just like we're using now, except that we need something that is left-aligned. Unfortunately, no such style exists, so we'll need to define one ourselves. Locate the "lblLarge" style and make a copy of it. Change the unique id to something appropriate (e.g. "lblLrgLeft") and change the alignment to "left". The result should look like the style below.

```
<style
  id="lblLrgLeft">
  <style_label
    textcolor="f0f0f0"
    font="fntLarge"
    alignment="left">
    </style_label>
  </style>
```

Once the style is defined, switch back to the template for arcane powers and locate the "name" portal. Change the style to the new one we just defined ("lblLrgLeft"). When you reload the data files, arcane powers should now have their name left-aligned properly.

Now we need a left-aligned solution for the currently centered "details" portal. Looking at the pre-defined styles in the file "styles_ui.aug" again, the "lblSmlLeft" style uses the "Small" font and is left-aligned. That's exactly what we need, so we'll make use of it within the appropriate portal. Locate the "details" portal and change the style to "lblSmlLeft". After a re-load, the details text should be properly left-aligned.

The last refinement we need to do for the template is to make sure our tab order is appropriate. The tab order of portals within a template is always the order in which the portals are actually defined in the template. As such, the portals should be sequenced so that the flow seems natural to the user. For example, the following sequence seems reasonable: "name", "details", "info", "delete", and "trappings". Revise the sequence as you deem fit and the template is now good to go.

Summary Information

The management of arcane powers is fully operational, but the "Arcane" tab should probably include some additional details about the selected arcane background, including the total number of powers and power points available to the character. We can present this in a template at the top of the panel, with the table appearing beneath the template. We'll keep the template simple, so we'll create

something that contains only a single, script-based label. All of the info we want to present will exist within the label, allowing the template height to be determined based on the height of the one portal. The net result is something that looks like the following.

```
<template
  id="apHeader"
  name="Arcane Header"
  compset="Actor">

  <portal
    id="info"
    style="lblNormal">
    <label>
      <labeltext><![CDATA[
       ~use a color that's is slightly less bright that the normal
text for powers
       @text = "{text c0c0c0}Total Powers: " &
#resmax[resPowers]

       ~insert a horizontal gap between the two values to space
them nicely
       @text &= "{horz 40} Total Power Points: " &
#trkmax[trkPower]
       ]]></labeltext>
     </label>
   </portal>

  <position><![CDATA[
    ~set up our height based on the tallest portal
    height = portal[info].height

   ~our portal should be centered in the template
    perform portal[info].centerhorz
    perform portal[info].centervert
    ]]></position>
  </template>
```

With the template in place, we can add the template to the layout. The template is positioned at the top, and the table consumes all remaining vertical space beneath. The result should look similar to the layout below.

```
<layout
  id="arcane">
  <portalref portal="apPowers" taborder="10"/>
  <templateref template="apHeader" thing="actor"/>
  <position><![CDATA[
   ~position and size the template and table to span the full
layout
    perform template[apHeader].autoplace
    perform portal[apPowers].autoplace[8]
    ]]></position>
  </layout>
```

DAMAGE (SAVAGE)

The next facet of the Savage Worlds game system that we're going to address is the management of damage. In Savage Worlds, damage works completely differently from the Skeleton files, so we're going to need to make substantial changes.

NEW MECHANICS

The damage mechanics for Savage Worlds are very simple, with characters suffering three wounds before being incapacitated. Fatigue works similarly, with there being two levels beyond zero before incapacitated. Characters must also track whether they are shaken, which is independent of the other two values. We could use a tracker for the wounds and fatigue, but the levels are so few and simple that it just doesn't make sense. Instead, we'll use a field to track the current value and use the normal health buttons for adjusting the damage level.

This results in the need for three new fields, which we'll add to the "Actor" component for simplicity. The "shaken" field must be designated as a "user" field, since the user will be responsible for toggling the state on/off. The other two fields will be controlled indirectly via up/down buttons. As such, they must both be declared as "derived" and must be assigned "full" persistence so that the values are saved and only change in response to the buttons. The three fields should look similar to the ones shown below.

```
<field
  id="acShaken"
  name="Is Shaken?"
  type="user">
  </field>

<field
  id="acWounds"
  name="Wounds"
  type="derived"
  persistence="full">
  </field>

<field
  id="acFatigue"
  name="Fatigue"
  type="derived"
  persistence="full">
  </field>
```

DELETE OLD MECHANICS

Since most game systems have a rather complex mechanic for damage, the Skeleton files provide support for that type of mechanic. However, none of that is needed for Savage Worlds, so we need to dispose of it all. The first item we need to delete is the "mscDamage" thing, which will be found in the file "thing_miscellaneous.dat". This particular thing is of zero use to us when managing damage for Savage Worlds.

The second task is to dispose of the "Damage" component and component set, which are both of no use to us. You'll find the component defined in the file "miscellaneous.str". The component set is auto-defined, since there is no need to blend other components into the component sets. Consequently, there is nothing to actually delete for the component set.

The third task is to dispose of all the health-related fields on the "Actor" component, except for the health summary ("acHPSumm"). These consist of the fields "acHPMin", "acHPMax", "acHPNow", and "acHPPenal".

Once the obsolete fields on the "Actor" component are gone, we need to adapt the health summary to the new mechanisms. The new logic should look something like the example below, where it properly reports the shaken state, wounds, and fatigue, highlighting any penalties appropriately.

```
<field
  id="acHPSumm"
  name="Health Summary"
  type="derived"
```

```
    maxfinal="50">
    <calculate phase="Render" priority="1000"><![CDATA[
     ~make sure this value consists of the elements that could
cause the summary to change
      @value = field[acShaken].value * 10000 +
field[acWounds].value * 100 + field[acFatigue].value
      ]]></calculate>

    <finalize><![CDATA[
      ~if we're not shaken and have incurred no negative effects,
all is good
      var net as number
      net = field[acWounds].value + field[acFatigue].value
      if (field[acShaken].value + net = 0) then
        @text = "-"
        done
        endif

      ~if we're shaken, signal it with a special indicator
      @text = ""
      if (field[acShaken].value <> 0) then
        @text &= "{font wingdings}v{revert}"
        endif

      ~if we've incurred wounds, report them
      var wounds as number
      wounds = -field[acWounds].value
      if (wounds <> 0) then
        if (empty(@text) = 0) then
          @text &= "/"
          endif
        if (wounds <= -4) then
          @text &= "Inc"
        else
          @text &= wounds & "W"
          endif
        endif

      ~if we've incurred fatigue, report it
      var fatigue as number
      fatigue = -field[acFatigue].value
      if (fatigue <> 0) then
        if (empty(@text) = 0) then
          @text &= "/"
          endif
        if (fatigue <= -3) then
          @text &= "Inc"
        else
          @text &= fatigue & "F"
          endif
        endif

      ~anything we output must be in red to highlight the penalty
      @text = "{text ff0000}" & @text
      ]]></finalize>
    </field>
```

The next step is to delete the usage pools associated with damage tracking. There are two of these ("DmgAdjust" and "DmgNet") and they will be found in the file "control.1st".

Once the usage pools are deleted, Eval script #1 within the "Actor" component will need to be revised to eliminate the reference to the usage pool. Change the script so that the test is based on the shaken state and the number of wounds and fatigue levels. The script below should serve nicely.

```
  <eval index="1" phase="Final" priority="1000"><![CDATA[
    ~if no damage has been incurred, assign a tag to indicate that
state
    if (field[acShaken].value + field[acWounds].value +
field[acFatigue].value = 0) then
      perform hero.assign[Hero.NoDamage]
    ~if the hero is dead or otherwise out of combat, indicate that
state
    elseif (field[acWounds].value >= 4) then
      perform hero.assign[Hero.Dead]
    elseif (field[acFatigue].value >= 3) then
      perform hero.assign[Hero.Dead]
      endif
    ]]></eval>
```

The final big task is to eliminate most of the portals on the "In-Play" tab that are associated with damage tracking. These will be found in the file "tab_inplay.dat" and will be located in the template "ipHealth". Since we'll still be wanting to adapt a few of these portals to the new mechanism, it's better to simply comment out the various portal elements instead of outright deleting them right now.

A few minor cleanup steps still remain. With all of the portals eliminated from the template, the Position script for the template needs to be omitted as well - we'll restore it and adapt it once we put some portals back in. The template itself needs to be switched over to the "Actor" component instead of the deleted "Damage" component. And lastly, the "templateref" within the layout needs to reference the "actor" thing instead of the deleted "mscDamage" thing.

At this point, you should now be able to re-compile the data files, although the "Health" section of the "In-Play" tab will not yet behave in any useful fashion.

MANAGING DAMAGE

We can now add damage management back into the data files in a systematic fashion. The first thing we need to add is a mechanism for the user to toggle the shaken state of the character. For now, we'll use a simple checkbox - we can always refine it later. In order to highlight the checkbox more prominently when a character is shaken, we'll modify the checkbox to display dynamically changing text, where the text changes to a bright red color when the character is shaken. This requires that we add another field to the "Actor" component wherein the dynamic text can be managed. The new field should look something like the following.

```
  <field
    id="acShakeTxt"
    name="Shaken Text"
    type="derived"
    maxfinal="50">

    <calculate phase="Render" priority="1000">
      @value = field[acShaken].value
      </calculate>

    <finalize><![CDATA[
      @text = ""
      if (@value <> 0) then
        @text = "{text ff0000}"
        endif
      @text &= "Shaken"
      ]]></finalize>
    </field>
```

The checkbox portal that we're adding to the template should look similar to the following.

```
  <portal
    id="shaken"
    style="chkLarge"
```

```
    tiptext="Click to indicate the character is shaken">
    <checkbox
     field="acShaken"
     dynamic="acShakeTxt">
     </checkbox>
    </portal>
```

In the above checkbox, we specified a new checkbox style that is not pre-defined in the file "styles_ui.aug". We need a checkbox that shows larger text so that the shaken state is more prominently visible. This entails us defining a new style, which in turn requires that we define a new font. The new style definition should be very similar to the one shown below.

```
<style
  id="chkLarge">
  <style_checkbox
   textcolor="clrnormal"
   font="fntchecklg">
   </style_checkbox>
  <resource
   id="fntchecklg">
   <font
    face="Arial"
    size="48"
    style="bold">
    </font>
   </resource>
  </style>
```

We also need to track wounds and fatigue levels here. Both of them work pretty much the same, except that the fatigue track transitions to an incapacitated state one step earlier than wounds, so we'll use the same approach for both. Each will have a label portal to identify the grouping (wounds vs. fatigue), a script-based label portal to show the actual value, an action portal to sustain one level of negative progression, and an action portal to restore a level. We'll use the damage and healing buttons for sustaining and restoring, and we'll display the actual value with color highlighting when non-zero. We'll also make sure to use a large font to make things highly visible to the user. To keep wounds and fatigue distinct, we'll visually group the portals together for each purpose. All of the additional portals (besides the "shaken" checkbox and the title) are presented below.

```
<portal
  id="lblwounds"
  style="lblLarge">
  <label
   text="Wounds:">
   </label>
  </portal>

<portal
  id="wounds"
  style="lblXLarge">
  <label>
   <labeltext><![CDATA[
    var wounds as number
    wounds = -field[acWounds].value
    if (wounds > -4) then @text = wounds
    else
     @text = "Inc"
     endif
    if (wounds <> 0) then
     @text = "{text ff0000}" & @text
     endif
    ]]></labeltext>
```

```
   </portal>

<portal
  id="wndsustain"
  style="actDamage"
  tiptext="Click here to sustain one wound.">
  <action
   action="trigger">
   <trigger><![CDATA[
    ~if we've reached our maximum, there's nothing left to do
    if (field[acWounds].value >= 4) then
     done
     endif
    ~sustain one new wound
    field[acWounds].value += 1
    ~any wound sustained automatically makes the character
shaken
    field[acShaken].value = 1
    ]]></trigger>
   </action>
  </portal>

<portal
  id="wndheal"
  style="actHeal"
  tiptext="Click here to heal one wound.">
  <action
   action="trigger">
   <trigger><![CDATA[
    ~if we've reached our limit, there's nothing left to do
    if (field[acWounds].value = 0) then
     done
     endif
    ~heal one wound
    field[acWounds].value -= 1
    ]]></trigger>
   </action>
  </portal>

<portal
  id="lblfatigue"
  style="lblLarge">
  <label
   text="Fatigue:">
   </label>
  </portal>

<portal
  id="fatigue"
  style="lblXLarge">
  <label>
   <labeltext><![CDATA[
    var fatigue as number
    fatigue = -field[acFatigue].value
    if (fatigue > -3) then
     @text = fatigue
    else
     @text = "Inc"
     endif
    if (fatigue <> 0) then
     @text = "{text ff0000}" & @text
    endif
    ]]></labeltext>
   </label>
  </portal>

<portal
  id="ftgsustain"
  style="actDamage"
  tiptext="Click here to sustain one level of fatigue.">
  <action
   action="trigger">
   <trigger><![CDATA[
    ~if we've reached our maximum, there's nothing left to do
    if (field[acFatigue].value >= 3) then
```

```
        done
        endif
       ~sustain one new fatigue level
       field[acFatigue].value += 1
       ]]></trigger>
     </action>
   </portal>

 <portal
   id="ftgheal"
   style="actHeal"
   tiptext="Click here to restore one fatigue level.">
   <action
    action="trigger">
    <trigger><![CDATA[
      ~if we've reached our limit, there's nothing left to do
      if (field[acFatigue].value = 0) then
        done
        endif
      ~restore one fatigue level
      field[acFatigue].value -= 1
      ]]></trigger>
    </action>
   </portal>
```

The Position script for the template that appropriately arranges everything should look something like the one presented below.

```
 ~set up our height based on our title, a gap, and our tallest
portal
  height = portal[title].height + portal[wounds].height + 8

 ~if this is a "sizing" calculation, we're done
  if (issizing <> 0) then
    done
    endif

 ~the title should span the full width
  portal[title].width = width

 ~position the tallest portal beneath the title
  perform portal[wounds].alignrel[ttob,title,8]

 ~center all non-text portals vertically on the wounds portal
  perform portal[shaken].centeron[vert,wounds]
  perform portal[wndsustain].centeron[vert,wounds]
  perform portal[wndheal].centeron[vert,wounds]
  perform portal[fatigue].centeron[vert,wounds]
  perform portal[ftgsustain].centeron[vert,wounds]
  perform portal[ftgheal].centeron[vert,wounds]

 ~align the smaller text portals to have the same baseline
  perform portal[lblwounds].alignrel[btob,wounds,-3]
  perform portal[lblfatigue].alignrel[btob,wounds,-3]

 ~position the shaken portal on the left
  portal[shaken].left = 15

 ~position the wound portals next to the shaken portal
  perform portal[lblwounds].alignrel[ltor,shaken,35]
  perform portal[wounds].alignrel[ltor,lblwounds,3]
  portal[wounds].width = 30

 ~position the sustain and heal portals next to the wounds
  perform portal[wndsustain].alignrel[ltor,wounds,6]
  perform portal[wndheal].alignrel[ltor,wndsustain,6]

 ~position the fatigue portals next to the wound portals
  perform portal[lblfatigue].alignrel[ltor,wndheal,35]
  perform portal[fatigue].alignrel[ltor,lblfatigue,3]
  portal[fatigue].width = 30
  perform portal[ftgsustain].alignrel[ltor,fatigue,6]
  perform portal[ftgheal].alignrel[ltor,ftgsustain,6]
```

RENAME THE "ACHPSUMM" FIELD

When we were overhauling everything above, we left the "acHPSumm" field in place and simply changed its behavior. However, that field is inappropriate for Savage Worlds, so we should change it now that everything is again stable. We'll change the id to "acDmgSumm". When we re-compile, three errors are reported, which we can chase down and correct quickly. The three errors occur in the files "form_static.dat", "form_dashboard.dat", and "form_taccon.dat". All we need is a quick search-and-replace to swap in the new unique id.

APPLYING DAMAGE TO TRAIT ROLLS

The final task we need to perform with regards to damage is to properly apply the resulting penalties to all trait rolls. Before we can do that, we must accrue the net penalty from both wounds and fatigue into a single value. This is most easily done by adding a new calculated field to the "Actor" component. The new field simply tallies the various values into one value that can be used whenever needed to adjust the trait rolls. Below is an example of what this field should look like.

```
 <field
    id="acNetPenal"
    name="Net Penalties"
    type="derived">
    <calculate phase="Traits" priority="5000">
      <before name="Derived trtFinal"/><![CDATA[
      @value = -1 * (field[acWounds].value +
field[acFatigue].value)
      ]]></calculate>
    </field>
```

The critical detail of the above field is its timing. We need to have the net penalties tallied up prior to the final calculation of derived traits, as well as before the final display text is generated for attributes and skills (to include the trait roll adjustments). The earlier of the two occurs at a timing of Traits/6000, so we need to pick something earlier. We'll use a timing of Traits/5000 out of convenience. Once the net penalty is properly calculated at a suitable timing, we can apply the penalty to the various trait rolls. For derived traits, the penalty needs to be added directly to the final calculated value. This is performed in the lone Eval script for the "Derived" component. By revising the script to simply add the penalty, we get a script that looks like the one below.

```
 <eval index="1" phase="Traits" priority="6000" name="Derived
trtFinal"><![CDATA[
    field[trtFinal].value = field[trtBonus].value +
field[trtInPlay].value +  herofield[acNetPenal].value
    ]]></eval>
```

For attributes and skills, the penalty is applied to the actual trait "roll" instead of the trait itself. The trait roll is determined as part of the process for synthesizing the "trtDisplay" field in an Eval script for the "Trait" component. When the bonus is tallied from the "trtRoll" and "trtNoStack" fields, the penalty should also be factored in, resulting in the appropriate net roll adjustment being calculated, as shown below.

```
 <eval index="4" phase="Render" priority="5000" name="Calc
trtDisplay">
    <after name="Calc trtFinal"/><![CDATA[
    ~if this is a derived trait, our display text is the final value
   if (tagis[component.Derived] <> 0) then
     field[trtDisplay].text = field[trtFinal].value
     done
     endif

   ~bound our final value including in-play adjustments
    var final as number
    final = field[trtFinal].value
    if (final < 2) then
      final = 2
    elseif (final > 6) then
      final = 6
     endif

   ~convert the final value for the trait to the proper die type for
display
    var dietype as number
    var display as string
    dietype = final * 2
    display = "d" & dietype

   ~if there are any bonuses or penalties on the roll, append
those the final result
    var bonus as number
    bonus = field[trtRoll].value + field[trtNoStack].value +
herofield[acNetPenal].value
    if (bonus <> 0) then
      display &= signed(bonus)
      endif

   ~put the final result into the proper field
    field[trtDisplay].text = display
    ]]></eval>
```

REVAMP THE "STATIC" FORM (SAVAGE)

The area across the top of the main HL window is referred to as the "static" form (or sometimes panel), and now is an excellent time to re-work its contents for Savage Worlds. The form contents are defined within the file "form_static.dat".

REPLACE CONTENT

The Skeleton data files provide a "static" form with some standard, basic elements that we'll revise and adapt. The character name should almost never need to be changed, and we can definitely leave it untouched for Savage Worlds. Since Savage Worlds also supports multiple races, we can also leave the race chooser in place. For the remaining elements, we need to determine what users will find most helpful to have visible for Savage Worlds. While there is no "right" answer, we're going to settle on showing the character's health, power points, and Bennies.

Looking at the contents of the "static" form within the data files, there are three visual elements (portals and templates) that comprise the layout. All of the information that we're going to be replacing is found within the "stActor" template, so we'll focus our attention there. For each piece of information, there in one portal involved: a dynamic label that incorporates both a identifying prefix and the character data. We'll follow the same logic, resulting in three portals. For the health and power points, we can display the summary fields that are already being synthesized on the "Actor" component. For the Bennies, we'll need to directly access the value we want to display. The net result is the following three portals.

```
  <portal
    id="health"
    style="lblStatic">
    <label>
      <labeltext><![CDATA[
        @text = "Health: " & herofield[acDmgSumm].text
        ]]></labeltext>
      </label>
    </portal>

  <portal
    id="power"
    style="lblStatic">
    <label>
      <labeltext><![CDATA[
        @text = "Power: " & herofield[acPPSumm].text
        ]]></labeltext>
      </label>
    </portal>

  <portal
    id="bennies"
   style="lblStatic">
    <label>
      <labeltext><![CDATA[
        @text = "Bennies: " &
hero.child[trkBennies].field[trkUser].text
        ]]></labeltext>
      </label>
    </portal>
```

POSITIONING EVERYTHING

The positioning of the various portals within the template is quick and painless. All of the portals use the same style, so they all have the same height characteristics. Consequently, we can ignore vertical alignment altogether. Along the horizontal axis, the portals simply line up, left to right, with suitable gaps between them. The final task we need to do is determine the dimensions of the template based on the contents within, so we simply determine the extent of the rightmost and bottommost portals. The only thing special we need to accommodate is that a character without an arcane background should not show any power points, so we need to hide it when necessary. This yields a Position script that looks something like the one below.

```
  ~position the health portal on the left  portal[health].left = 0

  ~position the power point portal horizontally or hide it if not
applicable  var x as number
  if (hero.tagis[Arcane.?] <> 0) then
    perform portal[power].alignrel[ltor,health,15]    x =
portal[power].right
  else
    portal[power].visible = 0
    x = portal[health].right
    endif

  ~position the Bennies portal horizontally
  portal[bennies].left = x + 15

  ~hide any portals that don't fully fit within the visible horizontal
space
  if (portal[power].right > width) then
    portal[power].visible = 0
    endif
  if (portal[bennies].right > width) then
    portal[bennies].visible = 0
    endif

  ~our height is the bottommost extent of our portals
```

```
height = portal[bennies].bottom
```

CHARACTER ADVANCEMENT

ADVANCEMENT SUPPORT (SAVAGE)

The Savage Worlds game system uses a character advancement system that requires each separate advancement be performed in a strict, formalized sequence. This is because different types of advancements are restricted based on the relative state of different traits (e.g. skill values relative to their linked attributes), and that state changes with each advancement. Sequential advancement logic can be somewhat complex to support, so the Skeleton files provide a built-in mechanism to orchestrate serialized advancement that can be readily adapted for use with most game systems that require the mechanism. This section outlines how that adaptation is achieved for Savage Worlds.

IMPORTANT! Before continuing with the rest of this section, you should be familiar with the basics of how advancement is handled within the Skeleton files.

ACCESSING LINKED ATTRIBUTES

A fundamental facet of the advancement logic is that each advance is actually a gizmo that contains the details of the advance within it as child picks. Selections that reside within a gizmo are not directly attached to the hero. However, the Kit provides a mechanism called displacement that allows specific selections within a gizmo to appear and behave as if they are directly attached to the hero.

The advancement logic makes use of displacement so that new skills and edges added to a character via an advance behave as if the user added them directly to the hero. But this only applies to new skills and edges. If the advance increases an existing skill or attribute, the selected trait is not displaced to the hero, as it already exists on the hero. This distinction is critical when working with the advancement logic, as it has important implications for linkages.

Linkages look for the linked pick within the same parent context as the reference pick. For example, when accessing the linked attribute for a skill on the hero, that attribute will be sought within the hero. However, if the skill is within the gizmo for an advance, accessing the attribute linkage will look for the attribute within the gizmo. This will obviously fail, since there is no attribute within the gizmo. This will result in a run-time error being reported.

The use of displacement makes this whole mechanism a bit more murky. If a skill within a gizmo is displaced to the hero, then that skill behaves as if it exists directly on the hero. As such, attempts to access the attribute linkage for a displaced skill will succeed, as it works exactly like a skill that is directly added to the hero. However, skills within a gizmo that are not displaced will fail to access the linkage and report the run-time error.

The way that advancements work, new skills (and edges) get displaced onto the hero. However, increases to existing skills (and attributes) are not displaced, so the skill selection lives solely within the context of the gizmo. This means that any script performed on a skill will be processed in two different contexts. Skills added directly to the hero (or displaced onto the hero) will be able to access their linked attribute, while skills within the gizmos will not. Any script that accesses the attribute linkage must verify whether the linkage is available before accessing it. This can be determined by simply verifying whether the effective container for the pick is the hero, which is achieved by the script logic shown below.

```
if (container.ishero <> 0) then
  ~the container is the hero, so accessing linkages will succeed
  endif
```

So this begs the question of what happens when the linkage isn't accessible. The answer is "nothing", and that's perfectly acceptable in this situation. The reason for this is that attribute linkages are only needed for new skills that get added to the hero (both directly and via displacement). Skills that are selected as increases don't need to actually perform normal handling for skills. Instead, all they need to do is apply the appropriate increase to the existing skill within the hero, and that does not require access to the attribute linkage.

This conditional handling of access to the attribute linkage is needed in the Eval script that retrieves the identity tag of the linked attribute into the skill. The revised version of the script should look similar to the one below.

```
<eval index="3" phase="Setup" priority="5000"><![CDATA[
  ~only access the linkage if the skill is directly on the hero; if
not, we assume
  ~it is within an advancement gizmo and no linkage will exist
there; we also don't
  ~need the linked attribute tag on advancement skills, so it's a
non-issue
  if (container.ishero <> 0) then
    perform linkage[attribute].pullidentity[Attribute]
    endif
  ]]></eval>
```

NOTE! The attribute linkage of skills should not be confused with the "basis" linkage used within advances. Whenever an existing pick is selected via a chooser, that pick is automatically tracked by HL as a "basis" pick, and accessing that pick is achieved via the "linkage[basis]" script transition. This mechanism ensures that any advance applied to an existing pick (e.g. an attribute or skill increase) can always identify that "basis" pick in order to apply adjustments to it. The "basis" linkage is a special type of linkage that is very different from normal linkages, but it is accessed via the "linkage[]" transition for syntactic and semantic consistency within scripts.

INCREASING SKILLS VIA ADVANCES

After assessing the different advancement options for Savage Worlds, the only ones that introduce new complexities are those pertaining to skills, and there are two issues to contend with. The first wrinkle is the distinction between skills that are less than their linked attribute and those that are not, as a different advance must apply to each. The easiest way to solve this is by having each skill identify its own nature during evaluation, with an appropriate tag being assigned. Since it's an either-or situation, only a single tag is needed, with its presence indicating one state and its absence indicating the other.

We define a "Helper.LessThan" tag and add an Eval script to the "Skill" component to make the appropriate determination. The script can be processed very late in the overall logic and should look similar to the one below. Since the attribute linkage must be accessed, we need to limit our processing to skills that are directly on the hero (or displaced onto the hero).

```
<eval index="4" phase="Final" priority="10000"><![CDATA[
```

```
    ~only access the linkage if the skill is directly on the hero; if
not, it is
    ~likely within an advancement gizmo and no linkage will exist
there; we also
    ~don't need the linked attribute tag on advancement skills, so
it's a non-issue
    if (container.ishero <> 0) then
      if (field[trtFinal].value < linkage[attribute].field[trtFinal].value)
then
        perform assign[Helper.LessThan]
        endif
      endif
    ]]></eval>
```

The second area of complexity is the advance that allows the character to increase two separate skills as part of the one advance. Doing this is possible, but it would require a fair amount of work and would go beyond the framework provided by the built-in advancement mechanism. So a simpler alternative would be preferable. Fortunately, this is easy to solve if we have the user increase the two skills separately via distinct advances. We can accomplish this by introducing a new advance that only increases a single skill and assigning it an advancement "cost" of only half an advance. This way, the user can select two of these advances for the standard cost of a normal advance. Since there is only one situation like this, we can safely introduce this new mechanism and have it be relatively intuitive for the user to work with. The specifics of this approach will be implemented later when we define the skill-related advances.

Identify Existing Skills

There are some advances that need to uniquely identify individual skills for appropriate handling. To support this, every skill must possess a suitable identity tag, and those tags must be forwarded up to the hero for reference. This entails defining the identity tag and forwarding it exactly the same way that we've already done a few times. The XML element below will define the identity tag behavior, and the script below will handle the forwarding. These must be added to the "Skill" component in the file "traits.str".

```
<identity group="Skill"/>

<eval index="1" phase="Setup" priority="5000"><![CDATA[
  perform forward[Skill.?]
  ]]></eval>
```

Don't Apply Creation Costs

With the changes made thus far, attributes, skills, and edges can now be added/increased via advances. These advances must not count towards the number of attribute and skill points that are spent on a given character, nor any starting edges that a character may be entitled to. This requires appropriate handling for each different type of trait.

For attributes, we only want to accrue attribute points when the user increases the attribute during character creation. All other attribute increases (e.g. via advances) must be ignored. This is achieved by ensuring that our attribute points only utilize the user- specified value for the attribute, as all advances increase the "bonus" field. Fortunately, we're already doing this, so no adjustments are required.

For skills, we need to address two separate situations. First, skill advances must be handled the same way as attributes, limiting the cost to the value assigned to the user-specified field for the skill. Just like with attributes, we're already doing this, so no adjustments are required. The second situation is for new skills that are added via advances - not just increased. For such skills, the entire skill needs to be ignored when tallying the skill point cost, else we'll treat the initial adding of the skill as one skill point. This requires that we modify the existing Eval script for tallying the points so that skills added via advances are ignored. The revised script below shows the change that must be made within the "Skill" component.

```
 <eval index="2" phase="Setup" priority="5000"><![CDATA[
   ~if this skill is not added directly to the hero (i.e. an advance),
skip it entirely
   if (origin.ishero = 0) then
     done
     endif

   ~the base value for skills is two, so we need to adjust by one
to get the proper cost
   hero.child[resSkill].field[resSpent].value += field[trtUser].value
– 1
   ]]></eval>
```

Edges are never increased via advances, but they are added via advances just like skills. As such, they need to be handled the same way that skills are handled. The edge needs to be ignored when tallying edge slots when it is added via an advance. This entails that same change as above for the "Edge" component, yielding the following revised script.

```
 <eval index="2" phase="Setup" priority="5000"><![CDATA[
   ~if this edge is not added directly to the hero (i.e. an
advance), skip it entirely
   if (origin.ishero = 0) then
     done
     endif

   ~consume another edge slot
   #resspent[resEdge] += 1
   ]]></eval>
```

Configuring Serialized Advancement

The logic for serialized advancement can be enabled and configured for a game system via the file "definition.def". Within this file, the "advancement" element encapsulates all the generalized details for serialized advancement. Enabling advancement is controlled via the "enable" attribute. You can also customize the terminology HL uses in conjunction with advancement via the "createterm" and "advanceterm" attributes, although the default terms are usually acceptable.

Within the "advancement" element, there is a child "canadvance" element that contains the CanAdvance script. This script allows you to perform verification before allowing the user to transition from the initial "creation" mode into "advancement" mode. For example, if you want to ensure that the user has allocated all the appropriate starting resources before switching to "advancement" mode, you can perform the proper checks in this script. If the script returns an empty "@message" special symbol, HL assumes all is well and allows the transition to occur. If any message text is generated by the script, failure is assumed and the message is displayed to the user as an explanation of what the problem is. For Savage Worlds, we need to verify that the user has properly allocated all attribute and skill points, as well as any starting edges and any rewards for starting hindrances.

Also within the "advancement" element is a child "transition" element that contains the Transition script. This script determines the message to be displayed to the user when the character transitions into and out of "advancement" mode. Normally, a simple reminder of the transition's implications and how to transition back is all that is needed, but you can customize the script to whatever extent you feel is appropriate. The default script provided by the Skeleton files presents a suitable notification that should work in most cases. For Savage Worlds, the default script works great.

Putting it all together, the "advancement" element for Savage Worlds should look similar to the example shown below. The mechanism is enabled, the terms are left in their default state, the Transition script is unchanged, and the CanAdvance script is tailored for the needs of a Savage Worlds character.

```
 <advancement
   enable="yes"
   <canadvance><![CDATA[
     var bullet as string
     bullet = "{bmp bullet_red}{horz 4}"
     @message = ""

    ~perform tests to assure all starting resources have been
assigned
     if (#resleft[resCP] <> 0) then
       @message = @message & bullet & "Character points must
be assigned for the character.{br}"
       endif

    if (#resleft[resAbility] <> 0) then
       @message = @message & bullet & "Ability slots must be
assigned for the character.{br}"
       endif
    ]]></canadvance>
  <transition><![CDATA[
     if (state.iscreate <> 0) then
       @message = "{b}{text ffff00}Creation Phase{text
010101}{/b}"
       @message &= "{br}{br}"
       @message &= "{align left}You have unlocked your
character, thereby exiting the Character Advancement phase
and moving back to the Character Creation phase. "
       @message &= "{br}{br}"
       @message &= "While unlocked, traits defined during
character creation can be adjusted, as long as those traits have
not yet been altered on the Advances tab. "
       @message &= "Traits that already have advancements
applied to them will remain locked unless those advancements
are deleted. "
       @message &= "{br}{br}"
       @message &= "Lock your character and remove any
advancements on a trait if you wish to   revise the rating that trait
was given during character creation. "
     else
       @message = "{b}{text ffff00}Advancement Phase{text
010101}{/b}"
       @message &= "{br}{br}"
       @message &= "{align left}You have locked your character
creation traits. "
       @message &= "{br}{br}"
       @message &= "By locking your character creation traits,
you have begun the Character Advancement phase of play. "
       @message &= "While locked, you cannot alter traits
defined during character creation. "
       @message &= "Use the Advances tab while the character
is locked to allocate advances to new  abilities or to increase
existing traits. "
       @message &= "{br}{br}"
       @message &= "Unlock the character to go back to the
Character Creation phase. "
       endif
```

```
    </advancement>
```

Adding Advancements (Savage)

All of the basic mechanics for advancement are now in place, so it's time to add the individual advancements for Savage Worlds.

Dynamic Tag Expressions

Each advance entails the selection of something slightly different by the user, yet the advancement mechanism uses a single chooser through which the user makes selections. Consequently, each advancement must properly dictate what can be selected, and this is achieved by using a dynamic tag expression within the chooser. Each advance possesses a field that contains the tag expression to be used, and it can be either hard-wired or synthesized on the fly via scripts, as necessary. This field must contain a valid tag expression that is applied against all things/picks to appropriately filter the list to the set of valid choices for the user.

For example, an advance that allows the user to add a new skill needs to specify a tag expression that identifies only the skills that have not yet been added to the character. Similarly, an advance that increases an existing skill needs a tag expression that identifies only those skills. By tailoring the tag expression properly, that same chooser can be re-configured for each advance to allow the user to select a completely distinct set of objects.

Handling Attribute Increases

Increasing an attribute is simple, as it can be readily adapted from the example provided within the Skeleton data files. First, we need to specify suitable action text. Next, we need to assign the "Advance.Increase" tag so that it is handled as an increase of an existing trait. The last step is to specify the appropriate tag expression for identifying valid attributes that can be increased. These consist of any attribute that is not hidden and not already at its maximum. The final result looks like the thing definition below.

```
 <thing
   id="advAttrib"
   name="Increase Attribute"
   compset="Advance"
   description="Increase an attribute by one die type.">
   <fieldval field="advAction" value="Boost Attribute"/>
   <fieldval field="advDynamic" value="component.Attribute &
!Helper.Maximum &  !Hide.Attribute"/>
   <fieldval field="advCost" value="1"/>
   <tag group="Advance" tag="Increase"/>
   <child entity="Advance">
    </child>
   </thing>
```

Handling New Edges

Adding new edges via an advance is a little more complicated. The action text is easy and we assign the "Advance.AddNew" tag so that it is handled as the addition of something new. In addition, we need to require that the user choose something to be added, so we must assign the "Advance.MustChoose" tag to the child gizmo. The final piece is the tag expression, which needs to be synthesized on- the-fly based on the list of edges that have already been added to the character. Specifically, the list of edges that is valid for selection consists of all edges that have not yet been added. Since the identity tags of all edges are forwarded to the hero, we can request a list of those ids from the hero and massage the list so that it can be used as

part of our tag expression. Once we append the list of precluded edges to the initial tag expression, we have a result that will allow the user to choose any edge that has not already been added. This yields an advance similar to the one below.

```
 <thing
   id="advEdge"
   name="Gain a New Edge"
   compset="Advance"    description="Select a new edge of your
choice.">
   <fieldval field="advAction" value="New Edge"/>
   <fieldval field="advDynamic" value="component.Edge"/>
   <fieldval field="advCost" value="1"/>
   <tag group="Advance" tag="AddNew"/>
   <eval index="1" phase="Render" priority="1000">
    <before name="Assign Dynamic Tagexpr"/><![CDATA[
    ~get the list of all edges on the hero and assemble it as a list
of precluded tags
     var tagexpr as string
     tagexpr = hero.tagids[Edge.?," &  !"]
     ~if there are any tags to exclude, append them to the
tagexpr appropriately
     if (empty(tagexpr) = 0) then
      field[advDynamic].text &= " &  !" & tagexpr
      endif
    ]]></eval>
   <child entity="Advance">
    <tag group="Advance" tag="MustChoose"/>
    </child>
   </thing>
```

Adding New Skills

The advance for adding new skills is very similar to the one for adding new edges. The key difference is the script that synthesizes the tag expression that controls which skills can be added by the user. We can't use the same technique as for edges, because there are some skills that can be added multiple times (e.g. "Knowledge"). Instead, we have to iterate through all of the skills that have been added to the hero and determine whether each skill is unique or not. If a skill is unique, it gets added to the list of tags that must be precluded from user selection, while non-unique skills can be added any number of times. The net result should look similar to the advance shown below.

```
 <thing
   id="advSkill"
   name="Gain a New Skill"
   compset="Advance"
   description="Select a new skill at a d4 rating.">
   <fieldval field="advCost" value="1"/>
   <fieldval field="advAction" value="New Skill"/>
   <fieldval field="advDynamic" value="component.Skill &
!Hide.Skill"/>
   <tag group="Advance" tag="AddNew"/>
    <eval value="1" phase="Render" priority="1000">
     <before name="Assign Dynamic Tagexpr"/><![CDATA[
     ~get the list of all unique skills on the hero and assemble it
as a list of precluded tags
     var tagexpr as string
     foreach pick in hero where "component.Skill &  !Hide.Skill"
      if (each.isunique <> 0) then
       tagexpr &= " &  !Skill." & each.idstring
       endif
      nexteach
     ~if there are any tags to exclude, append them to the
tagexpr appropriately
     if (empty(tagexpr) = 0) then
      field[advDynamic].text &= tagexpr
      endif
    <child entity="Advance">
     <tag group="Advance" tag="MustChoose"/>
     </child>
    </thing>
```

Handling Skill Increases

As mentioned above, we need two separate mechanisms for adding skill increases. One advance is used for increasing skills that are equal to or greater than the linked attribute, while the other is for skills that are less than the attribute. This allows us to apply a cost of one full advance for one type of increase and only a cost of a half-advance for the other type, which enables users to choose two of the latter advance for the price of any other advance. We've already instrumented each skill to possess a "Helper.LessThan" tag if it is less than the linked attribute value, so all we need to do is test against that tag. The only other details we have to make sure differ are the action text, description, and cost in terms of advancement slots. In the end, we have two separate advances that differ from each other in only those few ways, as shown below.

```
 <thing
   id="advBoost1"
   name="Increase Greater Skill"
   compset="Advance"
   description="Increase one skill by one die type that the skill is
equal to or higher than the linked attribute.">
   <fieldval field="advCost" value="1"/>
   <fieldval field="advAction" value="Boost Greater Skill"/>
   <fieldval field="advDynamic" value="component.Skill &
!Helper.Maximum &  !Hide.Skill   &  !Helper.LessThan"/>
   <tag group="Advance" tag="Increase"/>
    <child entity="Advance">
     <tag group="Advance" tag="MustChoose"/>
     </child>
   </thing>

 <thing
   id="advBoost2"    name="Increase Lesser Skill"
   compset="Advance"
   description="Increase one skill by one die type, provided that
the skill is currently less than its linked attribute.{br}{br}{b}{text
ffff00}Note!{text 010101}{/b} Select this option   {b}{i}twice{/i}{/b}
as a single advance, specifying a separate skill for each. This
selection   consumes only half an advance slot.">
   <fieldval field="advCost" value=".5"/>
   <fieldval field="advAction" value="Boost Lesser Skill"/>
   <fieldval field="advDynamic" value="component.Skill &
!Helper.Maximum &  !Hide.Skill &   Helper.LessThan"/>
   <tag group="Advance" tag="Increase"/>
    <child entity="Advance">
     <tag group="Advance" tag="MustChoose"/>
     </child>
   </thing>
```

Attribute Limits

All of our advances are now in place and should be working. However, we still have a special requirement regarding advances that we need to properly enforce. Attributes can only be increased once every rank. While this is probably a pretty standard rule that most gaming groups will adhere to, we can't make that assumption. So we have to allow users to break the rule, which means we need to utilize validation.

Solving this entails a rather crude, but effective, approach. We can use a "foreach" loop to iterate through all of the advances and tally the total number of attribute advances within each rank. If any rank

has more than one attribute advance within it, we can report an error to the user so that it can be rectified. We must process the advances in the exact order in which they were added to the character, which means we need to utilize the built-in "_CompSeq_" sort set.

Unfortunately, there's an interesting wrinkle. Since there are four advances per rank through 80 XP and two advances per rank thereafter, we need to differentiate the two progressions appropriately. This means an attribute advance is allowed only once out of every block of four advances during the first 16 advances, but only once out of every two advances after that. During our loop, we need to handle this distinction properly.

To enforce this check, we'll create a new thing that is intended solely for validation. That means we need to add it to the file "thing_validate.dat". Since the thing is performing checks on the overall hero, it can derive from the "Simple" component set. And since the thing needs to perform its checks on all actors, we assign it the "Helper.Bootstrap" tag to ensure it gets automatically applied to all heroes.

Most validation things possess only an EvalRule script, since they merely need to perform the validation test, and we could implement this thing the same way. However, we can make things a little simpler and more efficient by performing the validation in two stages. Each EvalRule script assumes the test is failed, so all the checks must be performed to ascertain whether each test is satisfied, but it's much easier for us to simply assume success and bail out if we ever determine that the requirements aren't satisfied. Consequently, the first stage is an Eval script that determines whether all the requirements are satisfied and assigns a tag to the hero if anything is failed. The second stage is the EvalRule that merely checks for the tag and reports the validation result accordingly. All we need to do is define a new "Hero.MultiAttr" tag in the file "tags.1st" and write the two scripts, which yields a new thing that looks like the following.

```
<thing
  id="valAdvance"
  name="Advances"
  compset="Simple">
  <tag group="Helper" tag="Bootstrap"/>
   <eval index="1" phase="Validate" priority="8000"><![CDATA[
    var index as number
    var total as number
    var is_check as number
    ~iterate through all advances in the order they were added to
the character
    foreach pick in hero where "component.Advance" sortas
_CompSeq_

     ~determine whether we need to perform an actual check at
this interval
      is_check = 0
      if (index % 4 = 0) then
        is_check = 1
      elseif (index % 2 = 0) then
        if (index > 16) then
          is_check = 1
          endif
        endif

     ~if we're supposed to check, do so, then reset the tally
count
      ~Note: If we find an error, assign a tag and there's nothing
more to do.
      if (is_check <> 0) then if (total > 1) then
          perform hero.assign[Hero.MultiAttr]
          endif
        total = 0 endif

      ~if this is an attribute increase, tally it
      if (eachpick.tagis[AdvanceId.advAttrib] <> 0) then
        total += 1
        endif

      ~increment our index based on our cost and continue
       ~Note: Using the cost allows us to properly identify the
transitions between
       ~ experience point levels where we need to perform our
check.
       index += eachpick.field[advCost].value
       nexteach

     ~perform a check on the final set of advances
     if (total > 1) then
       perform hero.assign[Hero.MultiAttr]
       endif
     ]]></eval>

   <evalrule index="1" phase="Validate" priority="9000"
      message="Multiple attributes increased within a rank"
summary="Multiple attributes"><![CDATA[
     ~if we have no instances of multiple advances within a level,
we're good
     if (hero.tagcount[Hero.MultiAttr] = 0) then
       @valid = 1
       done
       endif

     ~mark associated tab as invalid
     container.panelvalid[advances] = 0
     ]]></evalrule>
    </thing>
```

Revise Journal Tab (Savage)

We've got advancements in place, but advancements are dependent upon the accumulation of experience points (XP). The Skeleton files provide a reasonable starting point for tracking XP and finances, along with a suitable tab where the user can manage these details. However, we can probably tailor it better for Savage Worlds. So the next thing we'll tackle is the Journal tab.

Assess Each Journal Entry

The first thing we need to do is look at the individual journal entries provided by the Skeleton files and assess what needs to be added, deleted, or changed. Savage Worlds uses a rather common approach for handling money and XP, so we don't really need to track anything additional for each journal entry. Some games employ complex currencies with multiple types of coins that must be tracked independently, and those games will require some re-work of each journal entry to allow the user to properly manage those currencies. Fortunately, Savage Worlds isn't such a game, so we can avoid that work.

Assess the Header

While the individual journal entries are fine as they are, the header information at the top only shows the total cash on hand and total accrued XP. It would probably be helpful to the user to also show the character's current rank, since that has implications on the various edges and arcane powers that can be selected. We could even include the number of XP needed to reach the next rank, but each rank is an even 20 XP, so it's something a user can instantly determine and therefore makes little sense for us to add.

This will be the third time we need to convert the character rank from a numeric value to the corresponding name. So we really need to resolve this before we proceed with our changes to the header.

Converting Rank from Value to Text

We need to convert the character rank from the numeric value used to track it internally to a string for display. Typically, when a field is involved, this can be done by simply defining a Finalize script that performs the conversion. However, we also need to convert the rank requirement used within the pre-requisite test for minimum character rank, and the Finalize script won't work for that, so we need to devise a different approach. The technique we'll use is to write a procedure that will receive the rank value to be converted in one variable and place the result in a separate variable for use by the caller.

This procedure has an interesting wrinkle to it, though. This procedure needs to be able to be called from within both a Label script and a Validate script. The Label script starts with an initial context of a pick/thing, while the Validate script starts with an initial context of a container. As such, the two script types have nothing in common, so there is no procedure context we can use that the two can share. The solution is to define a procedure with a script type of "none". This results in a procedure that has no context whatsoever, which means there is no initial script context for the procedure (implied or otherwise). Since we only need to convert a numeric value to a string, there is no context required, so we can use this technique.

Open the file "procedures.dat", where we can now add the new procedure that we'll call "RankName". We can steal the logic out of the file "tab_basics.dat" for use within our procedure. The net result is something that should look like the definition below.

```
<procedure id="RankName" scripttype="none"><![CDATA[
  ~declare variables that are used to communicate with our
caller
  var rankvalue as number
  var ranktext as string
   ~map the value to the corresponding string
  if (rankvalue = 0) then
    ranktext = "Novice"
  elseif (rankvalue = 1) then
    ranktext = "Seasoned"
  elseif (rankvalue = 2) then
    ranktext = "Veteran"
  elseif (rankvalue = 3) then
    ranktext = "Heroic"
  elseif (rankvalue = 4) then
    ranktext = "Legendary"
  else
    ranktext = "Invalid Value: " & rankvalue
    endif
  ]]></procedure>
```

Once the procedure is in place, we need to go back and convert the two existing references to rank over to call the new procedure. We'll start with the Label script for showing the rank on the Basics tab. Open the file "tab_basics.dat" and locate the "baRank" portal. We change the Label script to call the "RankName" procedure to perform the conversion, which yields a new script that mirrors the one below.

```
var rankvalue as number
var ranktext as string
rankvalue = herofield[acRank].value
call RankName
@text = ranktext & " (" & hero.child[resXP].field[resMax].value
& " XP)"
```

Next, we'll revise the logic within the "MinRank" component. Open up the file "components.core" and locate the "MinRank" component, then find its "prereq" element. We replace the if/then/else block with a suitable call to the "RankName" procedure, resulting in a new Validate script that looks like the one below.

```
<prereq iserror="yes" message="Minimum Rank required.">
  <valid><![CDATA[
   var rankvalue as number
   var ranktext as string

   ~get the minimum rank required for the thing to be valid
   rankvalue = altthing.field[rnkMinRank].value

   ~if the minimum rank is satisfied, we're good to go
   if (herofield[acRank].value >= rankvalue) then
     @valid = 1
     done
     endif

   ~mark the panel as invalid
   altthing.linkvalid = 0

   ~synthesize an appropriate validation error message
   call RankName
   @message = ranktext & " rank required."
   ]]></valid>
  </prereq>
```

Revise the Header

Now that we've got the new "RankName" procedure in place, we can put it to use within the header of the Journal tab. Open the file "tab_journal.dat" and locate the "jrHeader" template, then find its "info" portal. This portal contains a Label script that synthesizes the text to be shown within the header. We'll amend it to add a new entry that shows the character's rank, yielding a revised script similar to the following.

```
var rankvalue as number
var ranktext as string
rankvalue = herofield[acRank].value
call RankName
@text = "{text a0a0a0}Total $: " & herofield[acCashNet].value
@text &= "{horz 40} Total XP: " & #resmax[resXP]
@text &= "{horz 40} Rank: " & ranktext
```

Gear and Equipment

Equipment (Savage)

The Savage Worlds data files are starting to come together nicely now. However, we still have a few significant tasks left to complete. The most obvious one at this point is that we haven't dealt with equipment yet. We'll start with equipment in general and simple gear in particular.

Assess Differences

As with previous stages in the development of our data files, the first step is to assess the differences between the Savage Worlds game system and the default mechanisms provided by the Skeleton files. In general, the Skeleton files provide the majority of what we'll

need. There are only a small number of core differences between what's provided and what we need. The sections below will address each of those areas.

TIME PERIOD

The first thing we must handle for Savage Worlds is the notion of a time period. The Savage Worlds rulebook presents a small number of basic time periods, and different assortments of gear are available within each of those time periods. Given that there are a growing number of Savage Settings supplements for the game, and since each setting can establish its own array of available gear, our solution should be something that can be readily expanded to support those settings. The Kit's "sources" mechanism is exactly what we need.

The sources mechanism allows an author to define an assortment of different configuration settings that the user can control via the "Configure Hero" form. Each source is presented to the user as a different setting that can be toggled on and off, and the sources can be organized into an hierarchical structure if you wish. We can then associate sources to things, making those things dependent on whether the user enables the corresponding source(s). We'll setup a "Time Period" source that is not user-selectable, and then we'll create a variety of user-selectable sources as children beneath it for each time period.

To add our sources, open the file "control.1st" and locate the "Optional" source that is pre-defined. Delete that source and replace it with the new sources that we'll be using. We assign an explicit sort order to each time period so that they are sorted chronologically when displayed instead of using the default alphabetical order. We also need to designate one source as the default. The net result should look similar to the source definitions shown below.

```
 <source
   id="TimePeriod"
   name="Time Period"
   selectable="no"
   description="Time periods and Savage Settings that govern
the availability of equipment">
   </source>

 <source
   id="TimeMedi"
   name="Medieval"
   parent="TimePeriod"
   sortorder="1"
   default="yes"
   description="Enables the availability of equipment from the
Medieval time period">
   </source>

 <source
   id="TimePowder"
   name="Black Powder"
   parent="TimePeriod"
  sortorder="2"
  description="Enables the availability of equipment from the
Black Powder time period">
   </source>

 <source
   id="TimeModern"
   name="Modern"
   parent="TimePeriod"
   sortorder="3"
   description="Enables the availability of equipment from the
Modern time period">
   </source>
 <source
   id="TimeFuture"
   name="Futuristic"
   parent="TimePeriod"
  sortorder="4"
   description="Enables the availability of equipment from the
Futuristic time period">
   </source>
```

Once the sources are defined, we can associate a thing with a particular source via the "usesource" element. This element specifies the unique id of the source to which the thing is linked. If a given thing should be available for multiple sources, such as Medieval hand weapons also being available during the Black Powder time period, then the thing should be assigned both sources. If either source is enabled by the user, the thing will be available for selection. For example, an Umbrella from the "Modern Items" list within the rulebook would be associated with the "Modern" time period much like the following example.

```
<thing
   id="eqUmbrella"
   name="Umbrella"
   compset="Equipment"
   description="Description goes here">
   <fieldval field="grCost" value="5"/>
   <fieldval field="grWeight" value="2"/>
   <usesource id="TimeModern"/>
   </thing>
```

MILITARY COST

A variety of equipment in Savage Worlds is assigned a cost of "Military", which indicates that the item cannot be purchased and is only available to military troops. There are two basic ways that we can support this behavior within our data files. The first option is to designate a special cost value that, when assigned to a piece of equipment, indicates the equipment should be treated as having a cost of "Military". If we use a value of zero, then we aren't able to have any piece of equipment that properly has a zero cost, so that's not a good idea. However, if we use a non-zero value, then we have to do special handling everywhere when buying and selling equipment to treat that special value as actually having a zero cost, and that's not a very good idea either. The other option is to leave the actual cost as zero for military equipment and assign a special tag to designate the equipment appropriately. This is a better approach, since the only thing we have to do specially is ensure that any piece of equipment with the special tag is displayed as having a cost of "Military".

We start by defining our new tag. Since this is a tag the user will be able to control via the Editor, we'll add it to the "User" tag group. To make things as obvious as possible, we'll simply name the tag "Military". Open the file "tags.1st" and locate the "User" tag group, then define the new tag. It should look similar to the one below.

```
 <value id="Military"/> <!-- Identifies a piece of equipment with a
cost of "Military" -->
```

Now that the tag is defined, we need to recognize it properly and display our cost. This is most easily accomplished by the introduction of a Finalize script for the "grCost" field of equipment. We need to allocate suitable space for the finalized value, and then

we simply check for the tag within the script. If the tag is present, the value is "Military", else the value is the numeric default. Open the file "equipment.str" and locate the "Gear" component, then find the "grCost" field. It seems that the field already has a Finalize script that formats the monetary cost appropriately. So all we need to do is revise the script to handle the special case of the tag before doing anything else. The net result should look similar to the script below.

```
 ~if we have a military cost, report that fact  if (tagis[User.Military]
<> 0) then
   @text = "Mil"
   done
   endif

 ~if this is a thing, we have to determine the value for use below;
if we
  ~have a non-zero lot cost, override our value based on the lot
cost, else
  ~use multiply the unit code by the lot size. See the Calculate
script
  ~comments for more details.  if (ispick = 0) then
   if (field[grLotCost].value > 0) then
     @value = field[grLotCost].value
   else
     @value = field[grCost].value * field[lotsize].value
     endif
   endif

 ~convert the cost to a value for display appropriately
  if (@value = 0) then
    @text = chr(150)
  else
    var moneyvalue as number
    var money as string
    moneyvalue = @value
    call Money
    @text = money
    endif
```

BASIC GEAR CATEGORIES

The final wrinkle in the way basic gear is handled for Savage Worlds is that the gear is broken up into categories for improved organization. In order to parallel this, we need to define a new tag group for tracking the type of gear. We can then define the appropriate types as tags within the group. Once defined, the appropriate tag can be assigned to each piece of gear.

For consistency with the rulebook, we want to sort gear by type, so we need to designate the tag group as having an "explicit" sequence behavior. Once we do that, we need to assign an "order" attribute to each tag so that HL knows the order in which to sequence the various groupings. We assign values that are in keeping with the rulebook.

In order to allow users to extend the set of gear types without having to modify the core files, we'll also designate the tag group as being "dynamic". Putting it all together yields a tag group specification similar to the one below, which must be defined within the file "tags.1st".

```
<group
  id="GearType"
  dynamic="yes"
  sequence="explicit">
  <value id="Custom" order="0"/>
  <value id="Adventure" name="Adventuring Gear" order="1"/>
  <value id="Clothing" order="2"/>
  <value id="Food" order="3"/>
  <value id="Computers" order="5"/>
  <value id="Surveil" name="Surveillance" order="6"/>
  <value id="Ammo" name="Ammunition" order="7"/>
  </group>
```

ADD BASIC GEAR

All the pieces are in place so that we can now define all of the basic gear within the Savage Worlds core rulebook. A couple of sample items are defined below for reference. You can either define the rest yourself or pull them out of the completed Savage Worlds data files that are included.

```
<thing
  id="eqBackpack"
  name="Backpack"
  compset="Equipment"
  description="Description goes here">
  <fieldval field="grCost" value="50"/>
  <fieldval field="grWeight" value="2"/>
  <usesource id="TimeMedi"/>
  <usesource id="TimePowder"/>
  <tag group="GearType" tag="Adventure"/>
  <tag group="thing" tag="holder"/>
  </thing>

<thing
  id="eqTorch"
  name="Torch"
  compset="Equipment"
  stacking="merge"
  description="Description goes here">
  <fieldval field="grCost" value="5"/>
  <fieldval field="grWeight" value="1"/>
  <usesource id="TimeMedi"/>
  <usesource id="TimePowder"/>
  <tag group="GearType" tag="Adventure"/>
  </thing>
```

REVISE GEAR TAB

All the gear is in place, so let's look at how it's added and managed by the user on the Gear tab. All of the mechanisms associated with the Gear tab will be found in the file "tab_gear.dat", and the Skeleton files provide a solid starting point for our needs, but we'll refine everything a little bit for Savage Worlds. If we go to add a piece of gear, the first thing we'll notice is that the description area on the right for the selected piece of gear is rather narrow for a meaningful description. We can widen that region via use of the "descwidth" attribute within the "table_dynamic" element of the table portal. The default width is 250 pixels, so try increasing it to 275 and 300 to see which you prefer.

The next thing we'll notice is that the various gear options are sorted alphabetically, but they are not grouped like they are within the rulebook. As we discussed earlier, all tables list their contents alphabetically by default, but we can customize that behavior if we wish via the use of a "sort set". Open the file "control.1st" and you'll find a variety of "sortset" elements near the bottom. We need to add a new one that will sort the gear first by its type/category and then by name within each category. Each sort key within a sort set can be either a field or a tag group, specified by its unique id, and the order of the sort keys dictates the ultimate ordering of the items. We want our gear to be sorted by the type and then the name, so we'll need two sort keys. Putting it all together yields a sort set that looks similar to the one shown below.

```
<sortset
  id="Gear"
  name="Gear By Type and Name">
  <sortkey isfield="no" id="GearType"/>
  <sortkey isfield="no" id="_Name_"/>
  </sortset>
```

Once the sort set is defined, we can reference it from the table portal so that all gear is shown to the user in the order dictated by the sort set. This is accomplished by adding the "choosesortset" attribute to the "table_dynamic" element of the table portal and specifying the id of our new sort set. With both the "descwidth" and "choosesortset" attributes added, the resulting "table_dynamic" element should look like below.

```
<table_dynamic
  component="Gear"
  showtemplate="grGrPick"
  choosetemplate="grGrThing"
  choosesortset="Gear"
  buytemplate="BuyCash"
  selltemplate="SellCash"
  descwidth="275">
```

There is one thing we've overlooked, though. If a thing does not possess a tag from a tag group specified in a sort set, that thing is sorted to the end of the list. This means that the three custom gear things are all sorted last now, but they should remain at the top of the list. The solution is to add a new "GearType" tag called "Custom" that is assigned an order value to put it first, after which the tag can be assigned to the three custom gear things. Once this is done, our gear is behaving the way we want it to behave.

Weapons (Savage)

Now that basic gear is in, this section will examine a more complex type of equipment: weapons. We'll continue with the same process we began in the previous chapter, assessing the differences between the Savage Worlds game system and the default mechanisms provided by the Skeleton data files.

Weapons in General

There is only one facet of weapons in general that is distinct from the default behaviors of the Skeleton files. Both hand weapons and ranged weapons can possess the "armor piercing" attribute, with a corresponding rating being specified. Since there is a rating involved, we need to add a new field to the "BaseWeapon" component in order to track the rating. You'll find the component in the file "equipment.str", and the new field should look similar to the one shown below.

```
<field
  id="wpPiercing
  name="Armor Piercing"
  type="static">
  </field>
```

Once the new field is added, we then need to include it in the description text for the weapon. All descriptions are routed through the assorted procedures in the file "procedures.dat", so open that file. Locate the "InfoWeapon" procedure, which synthesizes the generic details of all weapons. Insert the necessary lines of script code to include the armor piercing rating within the details. It's probably best to only include the info for weapons that possess a non-zero armor piercing rating, so the new code should look similar to the lines below.

```
~report the armor piercing rating of the weapon (if any)
 if (field[wpPiercing].value > 0) then
   iteminfo &= "Armor Piercing: " & field[wpPiercing].value &
"{br}"
   endif
```

The final thing we need to do is add the armor piercing rating to the special notes that are shown for weapons on the Armory tab. The notes to be displayed are synthesized into the "wpNotes" field for weapons via an Eval script within the "WeaponBase" component. Locate this script and add the appropriate new code to include the armor piercing rating at the start. The new code should look similar to the lines below.

```
~report the armor piercing rating of the weapon (if any)
 if (field[wpPiercing].value > 0) then
  if (empty(special) = 0) then
    special &= ", "
    endif
  special &= "AP " & field[wpPiercing].value
  endif
```

Minimum Strength Requirement

Weapons may be specified as having a minimum strength requirement. If a weapon is equipped when that requirement is not meant, using the weapon incurs a penalty of -1 for each step of shortfall. The Skeleton files provide some basic mechanisms for managing the strength requirement, but we must apply the penalty appropriately ourselves. Open the file "equipment.str" and locate the "WeaponBase" component. The first Eval script handles the minimum strength requirement logic, applying a penalty to the weapon and having a place to apply a more general penalty. Within Savage Worlds, there is no generalized penalty, so those lines can be deleted from the script. The penalty adjustment applied is currently a flat "-1", so we must change it to represent the shortfall. The net result is a revised Eval script that looks similar to the following script.

```
<eval index="1" phase="Final" priority="5000"><![CDATA[
  ~if the minimum strength is satisfied, there's nothing to do
   if (field[wpStrReq].value <= #trait[attrStr]) then
     done
     endif

  ~assign a tag to indicate the requirement isn't met
   perform assign[Helper.BadStrReq]

  ~apply any penalty required with the specific weapon
   if (#trait[attrStr] < field[wpStrReq].value) then
    field[wpPenalty].value = #trait[attrStr] - field[wpStrReq].value
    endif

  ~if not equipped, there's nothing more to do
   if (field[grIsEquip].value = 0) then
    done
    endif    ]]></eval>
```

Weapon Categories

In the same way we encountered for basic gear, weapons in Savage Worlds are grouped into categories or types. If we want our weapons to be displayed to the user in appropriate groupings to

match the rulebook, we'll need to manage these groupings. This entails the creation of a new tag group that we'll call "WeaponType" and for which we'll define tags for each grouping. This gets added to the file "tags.1st". We can then assign the appropriate tag to each weapon. We could potentially create separate tag groups for hand weapons and ranged weapons, but there is no real benefit in doing it, so we'll just use a single tag group for all weapons. As we did for basic gear, the tag group needs to be assigned the "explicit" sequencing behavior, and each tag needs to be assigned an appropriate "order" attribute to designate the proper sort order to match the rulebook. The net result is a tag group that looks like the following.

```
<group
  id="WeaponType"    sequence="explicit">
  <value id="MedBlades" name="Medieval Blades" order="1"/>
  <value id="MedAxes" name="Medieval Axes and Mauls"
order="2"/>
  <value id="MedPoles" name="Medieval Pole Arms"
order="3"/>
  <value id="ModMelee" name="Modern Melee" order="4"/>
  <value id="FutMelee" name="Futuristic Melee" order="5"/>
  <value id="MedRange" name="Medieval Ranged"
order="10"/>
  <value id="Black" name="Black Powder" order="11"/>
  <value id="ModPistol" name="Modern Pistol" order="12"/>
  <value id="ModSMG" name="Modern Submachine Gun"
order="13"/>
  <value id="ModShotgun" name="Modern Shotgun"
order="14"/>
  <value id="ModRifle" name="Modern Rifle" order="15"/>
  <value id="ModAssault" name="Modern Assault Rifle"
order="16"/>
  <value id="ModMachine" name="Modern Machine Gun"
order="17"/>
  <value id="FutRange" name="Futuristic Ranged" order="18"/>
  </group>
```

With the weapon type available, we need to put it to use. This will be done through a new sort set that parallels what we did for basic gear, and we'll call it "Weapon". The sort set first sorts on the weapon type and then on the name of the thing, which results in the following definition to be added to the file "control.1st".

```
<sortset
  id="Weapon"
  name="Weapon By Type and Name">
  <sortkey isfield="no" id="WeaponType"/>
  <sortkey isfield="no" id="_Name_"/>
  </sortset>
```

Hand Weapons

We'll focus on hand weapons first. The critical issue with hand weapons is the "weapon die". The damage for most hand weapons is based on the wielder's Strength plus a weapon die. The interesting wrinkle is that the weapon die cannot exceed the wielder's Strength die. If a weapon is listed with a damage of "Str+d8" and the wielder's Strength is "d6", the actual damage is downgraded to "Str+d6". In addition, none of the benefits of the weapon (e.g. Parry bonus) are conferred if the weapon die exceeds the Strength die.

To handle this, we're going to need to add a new field to track the weapon die. Each weapon will specify the appropriate weapon die, and then we'll use an Eval script to downgrade it if the wielder's Strength die is less. The actual damage we display can then be synthesized via another script, after everything has been resolved. We'll worry about how (and whether) to confer the weapon benefits in the next section.

There are also a few weapons that don't have a weapon die and do a fixed amount of damage. For example, the Bangstick does 3d6 damage. In this case, we must not adjust the weapon damage. To handle this, we'll need to distinguish a weapon with a weapon die from a weapon without one.

The simplest solution to all of this is to use a tag to indicate the weapon die for a given weapon. If no tag is assigned to the weapon, then the damage is fixed. Otherwise, we'll use the tag to determine the weapon die and adjust it as appropriate based on the Strength.

To handle the weapon die tags, we must define a new tag group in the file "tags.1st". The tag group needs to have separate tags for each of the different die types. We can allow the tag value to be the weapon die value we want to use internally (e.g. "2" for "d4", just like we do for attributes). However, we'll make sure the name of each tag is the proper die type for display. The resulting tag group should look like the following.

```
<group
  id="WeaponDie"
  name="Weapon Die-Type">
  <value id="2" name="d4"/>
  <value id="3" name="d6"/>
  <value id="4" name="d8"/>
  <value id="5" name="d10"/>
  <value id="6" name="d12"/>
  </group>
```

We can now add a new field in which we'll track the weapon die. As part of this field, we can define a Calculate script that interprets the tag and sets up the die type appropriately. Since the weapon die only applies to hand weapons, we'll add the handling to the "WeapMelee" component, which is in the file "equipment.str". The resulting field definition looks like below.

```
<field
  id="wpDie"
  name="Weapon Die"
  type="derived">
  <calculate phase="Initialize" priority="5000"><![CDATA[
    ~setup our die type based on any assigned tag
    @value = tagvalue[WeaponDie.?]
    ]]></calculate>
  </field>
```

The field is in place, so we can define the Eval script to handle the downgrading. Since we know that the weapon is failing it's strength requirement here, we can assign it a suitable tag to indicate that fact. We have to time script to occur after the Strength is resolved. If a downgrade occurs, we need to check that state before the derived traits like Parry are resolved. This means our script timing must occur right after the Strength is resolved. This yields the following Eval script.

```
<eval index="4" phase="Traits" priority="4000">
  <after name="Calc trtFinal"/><![CDATA[
  ~if the weapon die is greater than the strength die, downgrade
it
  if (field[wpDie].value > #trait[attrStr]) then
    field[wpDie].value = #trait[attrStr]
    perform assign[Helper.BadStrReq]
    endif
```

```
]]></eval>
```

The final piece we need to deal with is displaying the damage for a weapon. When we display a thing to the user, no Calculate or Eval scripts are evaluated. That means we have two choices for displaying weapons based on their die type. The first option is to duplicate the damage within the "wpDamage" field. That's going to be error prone, so it's a poor choice. The alternative is to use a Finalize script, since a Finalize script is always evaluated for a thing. Unfortunately, Finalize scripts can't be used on text-based fields, so we can't define one for the "wpDamage" field.

The solution is to define a new value-based field and put a Finalize script on it. Within the Finalize script, we need to handle the display of both things and picks. For a thing, the Calculate script on the "wpDie" field has not been performed, so we need to pull the value from the tag. Otherwise, we can pull the value from the (possibly downgraded) field.

Since we'll want to always use this field for showing damage to the user, it needs to exist for all weapons. That means it needs to be defined on the "WeaponBase" component. If we have a ranged weapon that doesn't include the "wpDie" field, we must be careful not to try accessing the non-existent field. We can safeguard against this by checking for the "component.WeapMelee" tag before accessing the weapon die logic.

The role of our new field will be solely for display of the final damage for a weapon, so we'll name it accordingly. Putting all this together results in the new field definition shown below.

```
<field
  id="wpShowDmg"
  name="Final Damage for Display"
  type="derived"    maxfinal="20">
  <!-- If we have a weapon die, synthesize for display, else use
wpDamage -->
  <finalize><![CDATA[
    ~use the weapon die to display the damage, else use the
explicit damage
    if (tagis[component.WeapMelee] + tagis[WeaponDie.?] < 2)
then
      @text = field[wpDamage].text
    else
      var die as number
      if (@ispick = 0) then
        die = tagvalue[WeaponDie.?] * 2
      else
        die = field[wpDie].value * 2
        endif
      @text = "Str+d" & die
      endif
    ]]></finalize>
  </field>
```

The handling of the weapon die is now in place. However, we'll need to remember to always use the "wpShowDmg" field for displaying weapon damage. The Skeleton files assume the "wpDamage" field is used, so we need to go through everywhere and change the reference where appropriate. The list is lengthy, so we won't go through each here. You can do a search for "wpDamage" within all the data files and replace it will "wpShowDmg" anywhere that it is being used for display or output of some sort. In general, the only exception would be the definitions of weapons themselves.

PARRY AND REACH

There are two remaining aspects of hand weapons that differ from what's provided by the Skeleton files. The first is the parry adjustment, and the second is the weapon's reach. As with how we handled armor piercing, both of these aspects are numeric ratings, so we'll add new fields to track both of them. Since these aspects only apply to hand weapons, we'll add them to the "WeapMelee" component. The new fields should look similar to the ones shown below.

```
<field
  id="wpParry"
  name="Parry Adjustment"
  type="static">
  </field>

<field
  id="wpReach"
  name="Reach Distance"
  type="static">
  </field>
```

With the tracking of the Parry adjustment, we need to apply that adjustment appropriately when the weapon is equipped. That entails defining a new Eval script for the component. All it needs to do is apply the proper adjustment to Parry prior to when the final Parry value is calculated. However, there is an important wrinkle. If the weapon die requirement is not satisfied, the Parry bonus is not applied. This means we also have to schedule our script after we downgrade the weapon die. Fortunately, there is a small window in which we can achieve this, resulting in the script below.

```
<eval index="5" phase="Traits" priority="5000">
  <before name="Derived trtFinal"/>
  <after name="Downgrade Weapon Die"/><![CDATA[
  if (field[grIsEquip].value <> 0) then
    ~negative adjustments are always applied
    if (field[wpParry].value < 0) then
      perform
#traitadjust[trParry,+,field[wpParry].value,"Equipped Weapon"]
    ~otherwise, adjustments are only applied when the strength
is satisfied
    elseif (tagis[Helper.BadStrReq] = 0) then
      perform
#traitadjust[trParry,+,field[wpParry].value,"Equipped Weapon"]
      endif
    endif
```

Again as we did for armor piercing, we need to add these ratings to the weapon description details. This can be done within the "InfoMelee" procedure within the file "procedures.dat". Since the rulebook omits these ratings unless they are non-zero, we'll do the same within the description text. It would be ideal if we can indicate to the user when these abilities don't apply for a weapon. We could accomplish that by highlighting the adjustments in red when the weapon does not satisfy the strength requirements. This yields code that should look similar to the following.

```
~report the parry adjustment of the weapon (if any)
if (field[wpParry].value <> 0) then
  iteminfo &= "Parry Adjustment: " &
signed(field[wpParry].value) & "{br}"
  endif

~report the reach of the weapon (if any)
if (field[wpReach].value > 0) then
  iteminfo &= "Reach: " & field[wpReach].value & "{br}"
```

```
  endif

 ~if the weapon fails its strength requirement, change color and
style
  if (empty(special) = 0) then
    if (tagis[Helper.BadStrReq] <> 0) then
      special = "{text a00000}{i}" & special & "{/i}{text 010101}"
      endif
    endif
```

Our last task is to add these two ratings to the special notes shown for weapons on the Armory tab. This can be done via the Eval script within the "WeapMelee" component that performs this function. Since the logic is basically the same, we can use the code from adding the armor piercing as a template. As we did above, we'll highlight the abilities if they aren't available due to insufficient Strength. The new lines of code should look like below.

```
 ~if the weapon fails its strength requirement, change color and
style
  if (tagis[Helper.BadStrReq] <> 0) then
    iteminfo &= "{text a00000}{i}"
   endif

 ~report the parry adjustment of the weapon (if any)
  if (field[wpParry].value <> 0) then
    if (empty(special) = 0) then
      special &= ", "
      endif
    special &= "Parry " & signed(field[wpParry].value)
    endif

 ~report the reach rating of the weapon (if any)
  if (field[wpReach].value <> 0) then
    if (empty(special) = 0) then
      special &= ", "
      endif
    special &= "Reach " & field[wpReach].value
    endif

 ~if the weapon fails its strength requirement, restore color and
style
  if (tagis[Helper.BadStrReq] <> 0) then
    iteminfo &= "{/i}{text 010101}"
    endif
```

Revised Strength Requirements

There's an important detail that we overlooked in getting the weapon die working. The way we handled things changes the way that minimum strength requirements are detected for hand weapons. As we saw a few moments ago, the Skeleton files provide basic handling that keys on the "wpStrReq" field value. The will continue to work for ranged weapons, but it no longer works for hand weapons, so we need to revise those mechanisms.

There are two separate instances on the "WeaponBase" component that deal with strength requirements. One is an Eval script and the other is a pre-requisite. Since the method is changing for hand weapons only, we have two options. We can move these existing scripts to the "WeapRange" component and then create new versions within the "WeapMelee" component. Alternately, we can adapt these scripts to handle both cases. The logic change is rather simple, so we'll choose the latter option.

Within the pre-requisite test, the only change we need to make is how we determine the minimum strength to be tested. For hand weapons, we use the weapon die if it's available. Otherwise, we use the the field value. We'll use the same test from earlier to determine which method to employ. The revised Validate script is shown below.

```
 <validate><![CDATA[
   ~if this is a pick, we're valid
   ~NOTE! We assume that equipping an item without the
strength just applies penalties.
   if (@ispick <> 0) then
     @valid = 1
     done
     endif

   ~get the minimum strength required for the weapon
    var minstr as number
   if (altthing.tagis[component.WeapMelee] +
altthing.tagis[WeaponDie.?] < 2) then
     minstr = altthing.field[wpStrReq].value
   else
     minstr = altthing.tagvalue[WeaponDie.?]
     endif

   ~verify whether we meet the strength requirement
   if (#trait[attrStr] >= minstr) then
     @valid = 1
     endif
   ]]></validate>
```

The same logic needs to be applied to the Eval script. The only other difference is that we only want to assign the "Helper.BadStrReq" tag if it's not already present. We assign that tag when we downgrade the weapon die so that we can detect it for disabled abilities like Parry. Assigning it again is rather silly (albeit harmless). The resulting Eval script should look like the one below.

```
 <eval index="1" phase="Final" priority="5000"><![CDATA[
   ~get the minimum strength required for the weapon
   var minstr as number
   if (tagis[component.WeapMelee] + tagis[WeaponDie.?] < 2)
then
     minstr = field[wpStrReq].value
   else
     minstr = tagvalue[WeaponDie.?]
     endif

   ~if the minimum strength is satisfied, there's nothing to do
   if (minstr <= #trait[attrStr]) then
     done
     endif

   ~assign a tag to indicate the requirement isn't met
    ~Note: This tag may already be assigned, so don't duplicte it.
   if (tagis[Helper.BadStrReq] = 0) then
     perform assign[Helper.BadStrReq]
     endif

   ~apply any penalty required with the specific weapon
    field[wpPenalty].value = #trait[attrStr] - field[wpStrReq].value

   ~if not equipped, there's nothing more to do
    if (field[grIsEquip].value = 0) then
     done
     endif
   ]]></eval>
```

At this point, our strength requirements logic should be operating properly.

Damage Bonus

There's one thing that we have forgotten to handle for hand weapons. The "Katana" has a damage of "Str+d6+2". If the "d6"

indicates the weapon die, then the extra "+2" is not able to be accommodated in any way. The "wpBonus" field represents an attack bonus - not damage - so we need something else.

To solve this, we can add a new "wpDmgBonus" field that tracks an independent damage bonus. The field definition should look like the following.

```
<field
  id="wpDmgBonus"
 name="Damage Bonus"
  type="derived">
  </field>
```

Once the field is defined, we can integrate it into the final damage displayed by modifying the Finalize script for the "wpShowDmg" field. We add the script code below to the damage.

```
~append any weapon damage bonus
 if (field[wpDmgBonus].value <> 0) then
   @text &= signed(field[wpDmgBonus].value)
 endif
```

ADD HAND WEAPONS

Everything is now in place for us to start adding all of the various hand weapons to the data files. All weapons should be added to the file "thing_armory.dat". A few examples are provided below. You can either add the rest or pull them out of the complete Savage Worlds data files that are provided.

```
<thing
  id="wpGreatswd"
  name="Greatsword"
  compset="Melee"
  description="">
  <fieldval field="wpParry" value="-1"/>
  <fieldval field="grCost" value="400"/>
  <fieldval field="gearWeight" value="12"/>
  <usesource source="TimeMedi"/>
  <usesource source="TimePowder"/>
  <tag group="WeaponDie" tag="5"/>
  <tag group="WeaponType" tag="MedBlades"/>
  <tag group="Equipment" tag="TwoHand"/>
  </thing>

<thing
  id="wpSpear"
  name="Spear"
  compset="Melee"
  description="">
  <fieldval field="wpParry" value="1"/>
  <fieldval field="wpReach" value="1"/>
  <fieldval field="grCost" value="100"/>
  <fieldval field="gearWeight" value="5"/>
  <usesource source="TimeMedi"/>
  <usesource source="TimePowder"/>
  <tag group="WeaponDie" tag="3"/>
  <tag group="WeaponType" tag="MedPoles"/>
  <tag group="Equipment" tag="TwoHand"/>
  </thing>

<thing
  id="wpSurvKnif"
  name="Survival Knife"
  compset="Melee"
  description="">
  <fieldval field="wpSpecial" value="Contains supplies that add
+1 to Survival rolls"/>
  <fieldval field="grCost" value="50"/>
  <usesource source="TimeModern"/>
  <tag group="WeaponDie" tag="2"/>
  <tag group="WeaponType" tag="ModMelee"/>
  <eval index="1" phase="PreTraits" priority="5000">
    <before name="Calc trtFinal"/><![CDATA[
    perform #traitroll[skSurvival,+,1,"Survival Knife"]
    ]]></eval>
  </thing>
```

While we're adding all of the hand weapons, we also need to revamp the provided "Unarmed Strike" weapon so that it behaves properly as a Savage Worlds weapon. The revised version should look similar to the following.

```
<pre>
 <thing
  id="wpUnarmed"
  name="Unarmed Strike"
  compset="Melee"
  description=""
  isunique="yes"
  holdable="no">
  <fieldval field="wpDamage" value="Str"/>
  <fieldval field="wpSpecial" value=""/>
  <fieldval field="grCost" value="0"/>
  <fieldval field="gearWeight" value="0"/>
  <tag group="Equipment" tag="Natural"/>
  </thing>
```

REVISE TABLE FOR HAND WEAPONS

All of the hand weapons are in place, so we can now revise how the weapons are managed visually within the table. The entirety of the "Amory" tab is defined within the file "tab_armory.dat", so that's where we'll be making our next changes. The table portal for hand weapons is named "arMelee".

We'll start with the same two changes we made to the table for showing basic gear. First, the width of the area for showing weapons descriptions is too narrow, so add the "descwidth" attribute to the "table_dynamic" element with a value of either 275 or 300. Second, we need to sort the weapons using the sort set we just defined for weapons. So add the "choosesortset" attribute to the same "table_dynamic" element and assign it the unique id of our new sort set: "Weapon".

The next thing we need to adjust is that the Skeleton files we inherited refer to "Melee Weapons", while Savage Worlds uses the term "Hand Weapons". Internally, this distinction is not important, but we want everything to look familiar to users. So we need to modify the scripts within the "headertitle" and "additem" elements to change the terms used.

After making the above changes, we should end up with a revised "arMelee" portal that looks similar to the one below.

```
<portal
  id="arMelee"    style="tblNormal">
  <table_dynamic
    component="Gear"
    showtemplate="arWpnPick"
    choosetemplate="arWpnThing"
    choosesortset="Weapon"
    buytemplate="BuyCash"
    selltemplate="SellCash"
    descwidth="275">
    <list>component.WeapMelee</list>
    <candidate inheritlist="yes">!Equipment.Natural</candidate>
```

```
    <titlebar><![CDATA[
      @text = "Select Hand Weapons to Purchase from the List
Below"
      ]]></titlebar>
    <headertitle><![CDATA[
      @text = "Hand Weapons"
      ]]></headertitle>
    <additem><![CDATA[
      @text = "Add New Hand Weapons"
      ]]></additem>
    </table_dynamic>
  </portal>
```

We should now ask ourselves if there are any of the new fields that we'd like to see shown for weapons that have been added to the character. The two solid candidates are the Parry adjustment and the AP rating for the weapon. Whenever a weapon is equipped, any Parry adjustment is automatically applied, so it's not something that the user can forget to add, which means it's really not important to remind the user about it. The AP rating, though, it quite different. It's very important that the user not forget about an AP rating for a wielded weapon, so we really should make a point of showing it.

Showing the AP rating within the template entails a number of additions and revisions. It all centers upon the template, which is named "arWpnPick" and will be found in the file "tab_armory.dat". Before we proceed, note that this one template is used for both hand weapons and ranged weapons, so our additions will appear for both weapon types unless we takes steps to only show it for one type. Since the AP rating is highly applicable for both weapon types, we'll just add it so that it's visible for all weapons.

The first thing we need to do is add a new portal to the template. Since we want the new portal to appear between the damage rating and any range information for the weapon, we'll position the new portal between those two portals within the template. If we use a field-based label portal, we'll only be able to display the raw value, which won't be very clear to the user. So we'll use a script-based label portal instead, allowing us to include a useful prefix and display the AP rating in the form "AP2". This means our new portal should look similar to the one below.

```
<portal
  id="piercing"    style="lblDisable">
  <label>
    <labeltext><![CDATA[
      if (field[wpPiercing].value > 0) then
        @text = "AP" & field[wpPiercing].value
      else
        @text = ""
        endif
      ]]></labeltext>
    </label>
  </portal>
```

Once the portal has been added, we now need to position it properly. In order to make room for it, we'll also need to nudge around some of the other portals. After a little bit of experimenting, a suitable result can be achieved, yielding the following new Position script.

```
~set up our height based on our tallest portal
 height = portal[info].height
~if this is a "sizing" calculation, we're done
 if (issizing <> 0) then
   done
   endif
~center the portals vertically
 perform portal[info].centervert
 perform portal[name].centervert
 perform portal[attack].centervert
 perform portal[damage].centervert
 perform portal[piercing].centervert
 perform portal[range].centervert
 perform portal[special].centervert
 perform portal[gearmanage].centervert
 perform portal[delete].centervert
 perform portal[strreq].centervert
 perform portal[heldby].centervert

~position the delete portal on the far right
 perform portal[delete].alignedge[right,0]

~position the info portal to the left of the delete button
 perform portal[info].alignrel[rtol,delete,-6]
~position the gear portal to the left of the info button
 perform portal[gearmanage].alignrel[rtol,info,-6]

~position the special portal to the left of the gear button
 perform portal[special].alignrel[rtol,gearmanage,-6]

~position the range portal to the left of the delete button; we
want the
  ~damage to be centered in its own column relative to a
centerpoint position
  perform portal[range].centerpoint[horz,340]

~position the AP portal to the left of the range column; we want
the AP to
  ~be centered in its own column relative to a centerpoint
position
  perform portal[piercing].centerpoint[horz,290]

~position the damage portal to the left of the AP column; we
want the damage
  ~to be centered in its own column relative to a centerpoint
position
  perform portal[damage].centerpoint[horz,240]

~position the attack portal to the left of damage column; we
want the
  ~attack to be centered in its own column relative to a
centerpoint position
  perform portal[attack].centerpoint[horz,190]

~position the name on the left and let it use all available space
  var limit as number
  limit = portal[attack].left - 8 - portal[heldby].width - 5 -
portal[strreq].width – 2
  portal[name].left = 0
  portal[name].width = minimum(portal[name].width,limit)

~show the 'strength requirement' icon to the right of the name
  perform portal[strreq].alignrel[ltor,name,2]
  portal[strreq].visible = tagis[Helper.BadStrReq]

~show the 'held by' icon to the right of the strenght requirement
if appropriate
  if (portal[strreq].visible = 0) then
    portal[heldby].left = portal[strreq].left
  else
    perform portal[heldby].alignrel[ltor,strreq,5]
    endif
  portal[heldby].visible = isgearheld

~if this is not a ranged weapon, hide the range portal
  if (tagis[component.WeapRange] = 0) then
    portal[range].visible = 0
    endif
```

```
~only show the special portal if there are special abilities/notes
to view
 portal[special].visible = 1 - field[wpNotes].isempty
```

RANGED WEAPONS (SAVAGE)

Now that hand weapons are in place, we'll continue with ranged weapons.

RANGED WEAPONS

Ranged weapons in Savage Worlds introduce four new variable ratings that need to be tracked, as well as a handful of special attributes that are either present or not. The variable ratings corresponding to the following characteristics: rate of fire, number of shots, number of actions to reload, and ammunition type. They can be managed via new fields that we'll add to the "WeapRange" component, which is defined in the file "equipment.str". They should look similar to the field definitions provided below.

```
<field
  id="wpReload"
 name="Actions to Reload"
  type="static">
  </field>

<field
  id="wpFireRate"
  name="Rate of Fire"
  type="static"
  defvalue="1"
  maxlength="10">
  </field>

<field
  id="wpShots"
  name="Number of Shots"
  type="static">
  </field>

<field
  id="wpAmmo"
  name="Ammunition"
  type="static"
  maxlength="10">
  </field>
```

The special attributes represent facets of weapons that are optionally present, such as snap fire, double tap, three-round burst, etc. All of these are either present or not, so they can be best represented as individual tags. The Skeleton files provide a tag group intended just for this purpose, with the id "Weapon". So we can add each of the tags we need to this tag group in the file "tags.1st". Whichever tags apply to a given weapon can simply be assigned to the thing. The revised tag group should look similar to the example shown below.

```
<group
  id="Weapon">
  <value id="SpecRange" name="Range is special"/>
  <value id="SnapFire" name="Snap Fire"/>
  <value id="DoubleTap" name="Double Tap"/>
  <value id="HvyWeapon" name="Heavy Weapon"/>
  <value id="HighExplos" name="High Explosive"/>
  <value id="ThreeRound" name="Three-Round Burst"/>
  <value id="NoMove" name="May Not Move"/>
  <value id="Revolver"/>
  <value id="SemiAuto" name="Semi-Auto"/>
  <value id="FullAuto" name="Full-Auto"/>
  <value id="MedBurst" name="Medium Burst Template"/>
  <value id="SmallBurst" name="Small Burst Template"/>
  <value id="LargeBurst" name="Large Burst Template"/>
  <value id="ConeTempl" name="Cone Template"/>
  <value id="Reload1" name="1 action to reload"/>
  </group>
```

Just like we did for hand weapons, the new fields and attributes need to be included within the weapon description details. The description for ranged weapons is handled by the "InfoRange" procedure within the file "procedures.dat". We only add the new information when it applies. This yields code that should look similar to the following.

```
<procedure id="InfoRange" context="info"><![CDATA[
  ~declare variables that are used to communicate with our
caller
  var iteminfo as string

  ~start with generic details for all weapons
  call InfoWeapon

  ~add the range details for the weapon
  iteminfo &= "Range: " & field[wpRange].text & "{br}"

  ~report the fire rate of the weapon (if any)
  iteminfo &= "Rate of Fire: " & field[wpFireRate].text & "{br}"

  ~report the number of shots for the weapon (if any)
  if (field[wpShots].value > 0) then
    iteminfo &= "# Shots: " & field[wpShots].value & "{br}"
    endif

  ~report the number of actions to reload the weapon (if any)
  if (field[wpReload].value > 0) then
    iteminfo &= "Actions to Reload: " & field[wpReload].value &
"{br}"
    endif

  ~report the ammunition for the weapon (if any)
  if (field[wpAmmo].isempty = 0) then
    iteminfo &= "Ammunition: " & field[wpAmmo].text & "{br}"
    endif
  ]]></procedure>
```

We also need to add the new fields and attributes to the special notes shown for weapons on the Armory tab. This can be done via the Eval script within the "WeapRange" component that performs this function. The exception is the ammunition type, which doesn't belong here and can be accessed via the description text if the user wants to reference that info. The code parallels what we've already done for hand weapons, which results in a revised script that looks like the following.

```
<eval index="2" phase="Render" priority="2000"><![CDATA[
  var special as string
   ~report the rate of fire for the weapon
  if (empty(special) = 0) then
    special &= ", "
    endif
  special &= "RoF " & field[wpFireRate].text

  ~report the number of shots for the weapon (if any)
  if (field[wpShots].value <> 0) then
    if (empty(special) = 0) then
      special &= ", "
      endif
    special &= "Shots " & field[wpShots].value
    endif
```

```
  ~report the reload actions of the weapon (if any)
  if (field[wpReload].value <> 0) then
    if (empty(special) = 0) then
      special &= ", "
      endif
    special &= "Reload " & field[wpReload].value & " action(s)"
    endif

  ~prepend any existing special details with the notes for this
weapon
  if (empty(special) = 0) then
    if (field[wpNotes].isempty = 0) then
      special &= ", "
      endif
    field[wpNotes].text = special & field[wpNotes].text
    endif
  ]]></eval>
```

ADD RANGED WEAPONS

We can now begin adding all of the various ranged weapons to the data files. As with hand weapons, they should be added to the file "thing_armory.dat". A few examples are provided below. You can either add the rest or pull them out of the complete Savage Worlds data files that are provided.

```
<thing
  id="wpCrossbow"
  name="Crossbow"
  compset="Ranged"
  description="Description goes here">
  <fieldval field="wpDamage" value="2d6"/>
  <fieldval field="wpShort" value="15"/>
  <fieldval field="wpMedium" value="30"/>
  <fieldval field="wpLong" value="60"/>
  <fieldval field="wpStrReq" value="3"/>
  <fieldval field="wpPiercing" value="2"/>
  <fieldval field="wpReload" value="1"/>
  <fieldval field="grCost" value="500"/>
  <fieldval field="grWeight" value="10"/>
  <usesource id="TimeMedi"/>
  <tag group="WeaponType" tag="MedRange"/>
  <tag group="Equipment" tag="TwoHand"/>
  </thing>

<thing
  id="wpColt1911"
  name="Colt 1911"
  compset="Ranged"
  description="Description goes here">
  <fieldval field="wpDamage" value="2d6+1"/>
  <fieldval field="wpShort" value="12"/>
  <fieldval field="wpMedium" value="24"/>
  <fieldval field="wpLong" value="48"/>
  <fieldval field="wpPiercing" value="1"/>
  <fieldval field="wpShots" value="7"/>
  <fieldval field="wpAmmo" value=".45"/>
  <fieldval field="grCost" value="200"/>
  <fieldval field="grWeight" value="4"/>
  <usesource id="TimeModern"/>
  <tag group="WeaponType" tag="ModPistol"/>
  <tag group="Weapon" tag="DoubleTap"/>
  <tag group="Weapon" tag="SemiAuto"/>
  </thing>

<thing
  id="wpBarrett"
  name="Barrett"
  compset="Ranged"
  description="Description goes here">
  <fieldval field="wpDamage" value="2d10"/>
  <fieldval field="wpShort" value="50"/>
  <fieldval field="wpLong" value="200"/>
  <fieldval field="wpStrReq" value="4"/>
  <fieldval field="wpPiercing" value="4"/>
  <fieldval field="wpShots" value="11"/>
  <fieldval field="wpAmmo" value=".50"/>
  <fieldval field="grCost" value="750"/>
  <fieldval field="grWeight" value="35"/>
  <usesource id="TimeModern"/>
  <tag group="WeaponType" tag="ModRifle"/>
  <tag group="Weapon" tag="SnapFire"/>
  <tag group="Weapon" tag="HvyWeapon"/>
  <tag group="Equipment" tag="TwoHand"/>
  </thing>
```

REVISE TABLE FOR RANGED WEAPONS

With all of the ranged weapons in place, we can now revise how they are managed visually within the table. The table portal for ranged weapons is named "arRange" within the file "tab_armory.dat". After a quick review, there are only two things that we need to change, and those are the same changes we already made to the "arMelee" table portal above. We first need to increase the width of the area for showing weapon descriptions, so we add the "descwidth" attribute to the "table_dynamic" element with a value of either 275 or 300. Then we add the "choosesortset" attribute to the same "table_dynamic" element and assign it the unique id of our custom sort set, which is "Weapon".

FINISHING OFF WEAPONS (SAVAGE)

We now need to reconcile the remaining facets of weapons, such as the proper calculation of attack values and the handling of ammunition.

WEAPON ATTACK CALCULATIONS

Early in the overall development process for these data files, we determined that Savage Worlds calculated the attack for weapons very differently from the default logic provided by the Skeleton files, so we disabled it. Now that we have all the various weapons in place, it's a perfect time to re-visit that and get it working correctly. The "wpNetAtk" field within the "BaseWeapon" component is provided for exactly this purpose. However, that field is value-based and it needs to be text-based so that it can display something like "d8+2". This means our first task is to change the "wpNetAtk" field to be text-based, which we can do by assigning it a "maxlength" attribute of "50". The revised field should look similar to the following.

```
<field
  id="wpNetAtk"
  name="Net Attack"
  type="derived"
  maxlength="50">
  </field>
```

ONE FIELD TO HARNESS

Three While we're mucking with fields, we might as well address another need. In order to independently synthesize an appropriately adjusted trait roll for the weapon attacks, those scripts will need to access the net bonus/penalty for the particular trait. Currently, that information is tracked across three separate fields, so we need to introduce a new field that calculates the net adjustment and can be readily accessed. We'll call our new field "trtNetRoll", and it must be generated after any penalties are applied due to the load limit, so

we'll base our timing on the script we defined earlier that performs that role. In the end, our new field should look a lot like the definition below.

```
<field
  id="trtNetRoll"
  name="Net Roll Bonus"
  type="derived">
  <calculate name="Calc trtNetRoll" phase="Traits"
priority="9000">
    <after name="Apply LoadLimit"/><![CDATA[
    @value = field[trtRoll].value + field[trtProf].value +
herofield[acNetPenal].value
    ]]></calculate>
  </field>
```

Re-Usable Procedure for Display

The logic we'll want to use when synthesizing the net attack for display will be very similar to the logic used to generate the final trait rolls for display within the "trtDisplay" field. Consequently, our next task should be to carve out that logic for re-use by putting it into a procedure. Once it's in a procedure, it can be used from both the current script that synthesizes "trtDisplay" and any other script that synthesizes "wpNetAtk".

Looking closely at the logic, there are two distinct values that must be passed into the procedure, with a string being returned for use by the caller. The two inbound values are the die type for the trait and any bonus/penalty to the roll. Based on these two inputs, our procedure can interpret the values correctly and synthesize the appropriate string for display to the user. The only thing special we need to do is ensure that the die type is properly bounded, just like is already done within the logic that synthesizes "trtDisplay". In the end, we should end up with a procedure that looks very similar to the one provided below.

```
<procedure id="FinalRoll" scripttype="none"><![CDATA[
  ~declare variables that are used to communicate with our
caller
  var finaldie as number
  var finalbonus as number
  var finaltext as string

  ~bound our final die type appropriately
  var final as number
  final = finaldie
  if (final < 2) then
    final = 2
  elseif (final > 6) then
    final = 6
    endif

  ~convert the final value for the trait to the proper die type for
display
  var dietype as number
  dietype = final * 2
  finaltext = "d" & dietype

  ~if there are any bonuses or penalties on the roll, append
those the final result
  if (finalbonus <> 0) then
    finaltext &= signed(finalbonus)
    endif
  ]]></procedure>
```

Once the procedure is in place, we need to revise the Eval script that synthesizes the "trtDisplay" field so that it uses the new procedure. The net result should closely parallel the revised script below.

```
<eval index="4" phase="Render" priority="5000" name="Calc
trtDisplay">
  <after name="Calc trtNetRoll"/>
  <after name="Calc trtFinal"/><![CDATA[
  ~if this is a derived trait, our display text is the final value
  if (tagis[component.Derived] <> 0) then
    field[trtDisplay].text = field[trtFinal].value
    done
    endif

  ~calculate the net bonuses and penalties for the roll
  var finalbonus as number
  finalbonus = field[trtNetRoll].value

  ~generate the appropriate results for display
  var finaldie as number
  var finaltext as string
  finaldie = field[trtFinal].value
  call FinalRoll

  ~put the final result into the proper field
  field[trtDisplay].text = finaltext
  ]]></eval>
```

Calculating the Net Attack

We've now got all of the pieces in place to be able to properly synthesize the "wpNetAtk" field for display to the user. There are two separate scripts in the file "equipment.str" that synthesize the "wpNetAtk" field - one for ranged weapons and another for hand weapons. We need two separate scripts because one is based on the "Shooting" skill, while the other is based on the "Fighting" skill.

We'll start by modifying the Eval script within the "WeapRange" component. The timing does not need to be touched, but we otherwise need to replace the script with revised logic that is similar to the synthesis of "trtDisplay" above and uses the "Shooting" skill. The key differences are two-fold. First, the bonus needs to incorporate both any bonus/penalty from the trait and any separate bonus/penalty from the weapon itself. Second, we need to handle the case where the character does not possess the needed skill. If not, then the die type will be zero due to the use of the "childfound." transition, which the procedure will bound as if it's a "d4". We must also apply the appropriate "-2" adjustment to the roll for being unskilled. The net result should yield a script that looks similar to the one below.

```
<eval index="1" phase="Final" priority="7000" name="Calc
wpNetAtk">
  <after name="Calc trtNetRoll"/>
  <after name="Calc trtFinal"/><![CDATA[
  ~start with the bonuses and penalties associated with the
weapon
  var finalbonus as number
  finalbonus = field[wpBonus].value + field[wpPenalty].value
  ~apply the appropriate adjustment for the associated skill
  if (hero.childexists[skShooting] = 0) then
    finalbonus -= 2
  else
    finalbonus += hero.child[skShooting].field[trtNetRoll].value
    endif
  ~get the proper die type for the associated skill
  var finaldie as number
  finaldie = hero.childfound[skShooting].field[trtFinal].value

  ~generate the appropriate results for display
```

```
  var finaltext as string
  call FinalRoll

 ~put the final result into the proper field
  field[wpNetAtk].text = finaltext
  ]]></eval>
```

We can now define a very similar Eval script for the "WeapMelee" component that synthesizes the net attack based on the "Fighting" skill. The existing script can have the comments removed from around it and can then have its contents replaced with the revised logic shown below.

```
 <eval value="2" phase="Final" priority="7000" name="Calc
wpNetAtk">
   <after name="Calc trtNetRoll"/>
   <after name="Calc trtFinal"/><![CDATA[
   ~start with the bonuses and penalties associated with the
weapon
   var finalbonus as number
   finalbonus = field[wpBonus].value + field[wpPenalty].value

  ~apply the appropriate adjustment for the associated skill
   if (hero.childexists[skFighting] = 0) then
     finalbonus -= 2
   else
     finalbonus += hero.child[skFighting].field[trtNetRoll].value
     endif

  ~get the proper die type for the associated skill
   var finaldie as number
   finaldie = hero.childfound[skFighting].field[trtFinal].value

  ~generate the appropriate results for display
   var finaltext as string
   call FinalRoll

  ~put the final result into the proper field
   field[wpNetAtk].text = finaltext
   ]]></eval>
```

Ammunition

The final aspect of weapons that we need to address is ammunition. All ammunition is actually purchased and managed as gear via the Gear tab, but we'll address it as part of the weapons. There really isn't anything to deal with for ammunition, except that we need to add a new tag for the new type of gear. Once that's done, it's just a matter of adding all of the appropriate entries for each type of ammunition. A couple of examples are provided below. You can either add the rest or pull them out of the complete Savage Worlds data files that are provided.

```
<thing
   id="eqBulletSm"
   name="Bullets (Small)"
   compset="Ammunition"
   lotsize="50"
   stacking="merge"
   description="Includes .22 to .32 caliber weapons">
   <fieldval field="grCost" value=".2"/>
   <fieldval field="grWeight" value=".06"/>
   <tag group="GearType" tag="Ammo"/>
   </thing>

<thing
   id="eqLsrBatt"
   name="Laser Battery"
   compset="Ammunition"
   stacking="merge"
   description="Provides one full load of shots for the laser pistol,
rifle, or MG">
   <fieldval field="grCost" value="25"/>
   <fieldval field="grWeight" value="1"/>
   <usesource id="TimeFuture"/>
   <tag group="GearType" tag="Ammo"/>
   </thing>
```

Thrown Weapons

After doing a bit of testing of our changes, there is something we've overlooked. Savage Worlds has separate skills for "Shooting" and "Throwing". However, we neglected to distinguish that detail when calculating the net attack for ranged weapons.

The first thing we need to do is differentiate between thrown weapons and fired weapons. We can accomplish that by defining a new tag in the "Weapon" tag group in the file "tags.1st". This new tag will identify a weapon that uses the "Throwing" skill, with any weapon lacking the tag using the "Shooting" skill. The new tag definition is as simple as below.

```
<value id="Thrown"/>
```

With the tag in place, we can assign it to the appropriate weapons, such as a throwing knife. We can then modify the Eval script that calculates the "wpNetAtk" field within the "WeapRange" component. The overall logic remains the same, but we need to use the different skill when a weapon is thrown. The revised Eval script is shown below.

```
 ~start with the bonuses and penalties associated with the
weapon
  var finalbonus as number
  finalbonus = field[wpBonus].value + field[wpPenalty].value

 ~apply the appropriate adjustment for the associated skill
  if (tagis[Weapon.Thrown] <> 0) then
    if (hero.childexists[skThrowing] = 0) then
      finalbonus -= 2
    else
      finalbonus += hero.child[skThrowing].field[trtNetRoll].value
      endif
  else
    if (hero.childexists[skShooting] = 0) then
      finalbonus -= 2
    else
      finalbonus += hero.child[skShooting].field[trtNetRoll].value
      endif
    endif

 ~get the proper die type for the associated skill
  var finaldie as number
  if (tagis[Weapon.Thrown] <> 0) then
    finaldie = hero.childfound[skThrowing].field[trtFinal].value
  else
    finaldie = hero.childfound[skShooting].field[trtFinal].value
    endif

 ~generate the appropriate results for display
  var finaltext as string
  call FinalRoll

 ~put the final result into the proper field
  field[wpNetAtk].text = finaltext
```

ARMOR (SAVAGE)

Now that weapons are operational, we might as well switch our focus to armor and shields.

ARMOR CATEGORIES

We need to setup the handling for appropriate armor categories, just like we did for weapons. We'll define a new tag group for this purpose called "ArmorType", complete with tags for each category. As before, we designate the tag group as having "explicit" sequencing behavior, with each tag specifying its respective order. The net result is a tag group that looks like below, which is added to the file "tags.1st".

```
<group
  id="ArmorType"
  dynamic="yes"
  sequence="explicit">
  <value id="MedArmor" name="Medieval Armor" order="1"/>
  <value id="MedShield" name="Medieval Shields" order="2"/>
  <value id="ModArmor" name="Modern Armor" order="3"/>
  <value id="FutArmor" name="Futuristic Armor" order="4"/>
  </group>
```

We can now define a new sort set that relies upon our new tag group, and we'll call it "Defense". As before, the sort set first sorts on the armor type and then on the name of the thing, which results in the following definition in the file "control.1st".

```
<sortset
  id="Defense"
  name="Defensive Gear By Type and Name">
  <sortkey isfield="no" id="ArmorType"/>
  <sortkey isfield="no" id="_Name_"/>
  </sortset>
```

DEFENSIVE BONUS

Both armor and shields confer a defensive bonus to the character, although their use is distinct in Savage Worlds. However, we can define a single field on the "Defense" component to track this value and simply use it in different ways for armor and shields. The Skeleton files provide a "defDefense" field for us already, so we'll use that field for our purposes.

The Savage Worlds rulebook always shows the armor rating in the format "+X", so we should make sure that we do that same. This can be achieved by adding a Finalize script to the "defDefense" field that prefixes it with a "+" character for display. The revised field should look like the one shown below.

```
<field
  id="defDefense"
  name="Defense Adjustment"
  type="derived"    maxfinal="10">
  <finalize><![CDATA[
    @text = "+" & @value
    ]]></finalize>
  </field>
```

With the conversion of this field to possess a Finalize script, we need to realize that any references to this field within scripts that access its "value" will not get the newly synthesize text. So we need to do a scan of our data files and review any references to the "defDefense" field. Any reference to the field by value that is used in output for the user to see should be revised to access the field as text. This amounts to changing references from "field[defDefense].value" to "field[defDefense].text". Fortunately, after a quick scan, there don't appear to be any such instances we need to worry about.

The defensive bonus for armor needs to be added to the character's Toughness trait. However, we still need to handle the non- stacking nature of armor and the multiple zones of coverage before we can do this correctly. So we'll deal with this detail in a little bit.

FOUR ZONES OF COVERAGE

Armor protection within Savage Worlds is tracked separately for four different zones on a character: torso, head, arms, and legs. However, the armor protection conferred by a piece of equipment always has the same value, so it makes more sense to track a single value for the armor and then designate which zones are protected by the equipment. To support this, we'll define a new tag group called "ArmorLoc" that possesses four tags - one for each zone of protection. Each piece of armor can then be assigned one or more tags to indicate which zones are actually protected. This yields a new tag group that looks like below.

```
<group
  id="ArmorLoc">
  <value id="Torso"/>
  <value id="Head"/>
  <value id="Arms"/>
  <value id="Legs"/>
  </group>
```

NON-STACKING PROTECTION

Armor protection does not stack in Savage Worlds, so we need to identify the best protection in each zone of coverage for the character. The easiest way of resolving this is to create four new fields directly on the actor pick. These fields will track the best protection in each location. The new fields must be added to the "Actor" component, which is defined in the file "actor.str". They should look similar to those shown below.

```
<field
  id="acDefTorso"
  name="Armor on Torso"
  type="derived">
  </field>

<field
  id="acDefHead"
  name="Armor on Head"
  type="derived">
  </field>

<field
  id="acDefArms"
  name="Armor on Arms"
  type="derived">
  </field>

<field
  id="acDefLegs"
  name="Armor on Legs"
  type="derived">
  </field>
```

Once the new fields are defined, the next thing we need to do is put the appropriate values into them. An Eval script on the "Armor" component applies the effects of equipped armor, so we need to

revise that script. The script currently adds the defensive bonus of the armor to the character's Toughness. Instead of doing that, the script should assign the defensive bonus to the new actor field(s), doing so only if the new value is better than the current value and only for the locations covered by the armor. The new script should look similar to the one below.

```
<eval index="2" phase="PreTraits" priority="5000">
  <before name="Calc trtFinal"/>
  <after name="Equipped"/><![CDATA[
  ~if this gear is not equipped, skip it
  if (field[grIsEquip].value = 0) then
    done
    endif

  ~apply the appropriate trait adjustments for the equipment
  var defense as number
  defense = field[defDefense].value
  if (tagis[ArmorLoc.Torso] <> 0) then
    if (defense > herofield[acDefTorso].value) then
      herofield[acDefTorso].value = defense
      endif
    endif
  if (tagis[ArmorLoc.Head] <> 0) then
    if (defense > herofield[acDefHead].value) then
      herofield[acDefHead].value = defense
      endif
    endif
  if (tagis[ArmorLoc.Arms] <> 0) then
    if (defense > herofield[acDefArms].value) then
      herofield[acDefArms].value = defense
      endif
    endif
  if (tagis[ArmorLoc.Legs] <> 0) then
    if (defense > herofield[acDefLegs].value) then
     herofield[acDefLegs].value = defense
      endif
    endif
  ]]></eval>
```

As we test our changes, we'll find that we inherited a validation test from the Skeleton data files. This test verifies that the character doesn't try wearing more than one piece of armor at a time. Since Savage Worlds allows for multiple armor pieces with the non-stacking logic we just implemented, this validation test is not appropriate for us. The validation is performed via the "valArmor" thing, which resides in the file "thing_validate.dat". All we need to do is delete the thing and the erroneous validation is gone.

Unfortunately, that didn't fix the problem entirely. Although the validation error is no longer appearing, multiple pieces of equipped armor are still showing up in red. This is due to an Eval Rule script on the "Armor" component. Once we delete the script, the user is free to equip armor in whatever way he chooses.

TORSO ARMOR BOOSTS TOUGHNESS

At this point, we should now be tracking the best armor protection in each location. So we can now determine the protection afforded to the torso and add that to the character's Toughness trait. Since the best protection is tracked on the "Actor" component, we can define a new Eval script on that component to confer the appropriate protection to the Toughness trait. The new script must occur **after** we've accrued the best protection information, and it should end up looking like the following.

```
<eval index="4" phase="PreTraits" priority="6000">
  <before name="Calc trtFinal"/><![CDATA[
  #traitbonus[trTough] += field[acDefTorso].value
  ]]></eval>
```

BALLISTIC DEFENSE

Some kinds of armor (e.g. Kevlar Vest) possess a separate defense against bullets. This defense applies only against bullets and also negates some portion of the AP rating of the bullet. We should probably track this separate defensive rating on the "Armor" component. However, we're ultimately going to want to emulate the Savage Worlds rulebook for how to display this information. Since the rulebook incorporates this ballistic defense into the actual armor protection value shown (e.g. "+2/+4"), we want to incorporate it into the net value shown. The value shown is synthesized via a Finalize script for the "defDefense" field, which resides on the "Defense" component, so we must also put the new field on the "Defense" component. The new field should look similar to the following.

```
<field
  id="defBullets"
  name="Bullets Defense Adjustment"
  type="static">
  </field>
```

Now that the field is defined, we need to incorporate the value into the display of the "defDefense" field. This means revising the Finalize script to add any "defBullets" field that may exist, which means we also may need to allow a little more storage for the result. The revised field definition should look similar to what's shown below.

```
<field
  id="defDefense"
  name="Defense Adjustment"
  type="static"
  maxfinal="20">
  <finalize><![CDATA[
    @text = "+" & @value
    if (field[defBullets].value > 0) then
      @text &= "/+" & field[defBullets].value
      endif
    ]]></finalize>
  </field>
```

Shields

There are two facets of shields that need to be managed by the data files. First of all, some shields have an armor bonus, which is applied to ranged attacks that hit the character. We'll use the "defDefense" field for this purpose.

Shields also have a Parry bonus that must be applied when the shield is equipped. To track the Parry bonus, we need to add a new field to the "Shield" component. The field is a simple, static value, and it should look like the following.

```
<field
  id="defParry"
 name="Parry Adjustment"
  type="static">
  </field>
```

Once the field is defined, we need to apply the Parry bonus to the trait appropriately. This requires that we define a new Eval script within the "Shield" component that performs the task. If a shield is

equipped, it needs to add its Parry bonus to the Parry trait, resulting in the script below.

```
<eval index="2" phase="PreTraits" priority="5000">
  <before name="Calc trtFinal"/><![CDATA[
  if (field[grIsEquip].value > 0) then
    #traitbonus[trParry] += field[defParry].value
    endif
  ]]></eval>
```

Updated Description

Armor and shields both introduce a few facets that need to be included in the description text that is synthesized for display to the user. This text is generated within the "InfoDef" procedure that is found in the file "procedures.dat". For armor, we need to report what areas are covered. For shields, we need to report both the armor bonus that is applied to ranged attacks and the Parry bonus. We can differentiate these two object types via the "component" tag each possesses. This yields the following revised procedure for showing the new information.

```
<procedure id="InfoDef" context="info"><![CDATA[
  ~declare variables that are used to communicate with our
caller
  var iteminfo as string

  ~if this is armor, output the appropriate details
  if (tagis[component.Armor] <> 0) then
    ~add the armor rating
    iteminfo &= "Armor: " & field[defDefense].text & "{br}"
    ~identify the areas covered by the armor
    iteminfo &= "Coverage: " & tagnames[ArmorLoc.?,", "] &
"{br}"

  ~otherwise, this is a shield, so output the appropriate details
  else
    ~add the parry bonus
    iteminfo &= "Parry Bonus: " & signed(field[defParry].value) &
"{br}"
    ~add the ranged attack defense (if any)
    if (field[defDefense].value > 0) then
      iteminfo &= "Ranged Defense: " & field[defDefense].text &
"{br}"
      endif
    endif

  ~report the minimum strength requirement (omitting if there is
none)
  if (field[defStrReq].value > 0) then
    iteminfo &= "Minimum Strength: " & field[defStrReq].value &
"{br}"
    endif
  ]]></procedure>
```

Add Armor and Shields

All of the mechanics for armor and shields is now in place, so we can begin adding all of the various items to the data files. Just like weapons, they should be added to the file "thing_armory.dat". A few examples are provided below. You can either add the rest or pull them out of the complete Savage Worlds data files that are provided.

```
<thing
  id="armLeather"
  name="Leather Armor"
  compset="Armor"
  description="Description goes here">
  <fieldval field="grCost" value="50"/>
  <fieldval field="gearweight" value="15"/>
  <usesource id="TimeMedi"/>
  <tag group="ArmorType" tag="MedArmor"/>
  <tag group="ArmorLoc" tag="Torso"/>
  <tag group="ArmorLoc" tag="Arms"/>
  <tag group="ArmorLoc" tag="Legs"/>
  </thing>

<thing
  id="shdMedium"
  name="Medium Shield"
  compset="Shield"
  description="Description goes here">
  <fieldval field="defDefense" value="2"/>
  <fieldval field="defParry" value="1"/>
  <fieldval field="grCost" value="50"/>
  <fieldval field="gearweight" value="12"/>
  <usesource id="TimeMedi"/>
  <tag group="ArmorType" tag="MedShield"/>
  </thing>
```

Revise Table for Armor and Shields

The final task is to revise the way that armor and shields are visually managed within the UI. There is a table portal for defensive equipment that is named "arDefense" within the file "tab_armory.dat". After reviewing how everything is behaving, there are only two details that need to be changed, and they are the same two things that we needed to change for weapons. First, we need to increase the width of the area for showing armor descriptions, which entails adding the "descwidth" attribute to the "table_dynamic" element with a value of either 275 to 300. Second, we add the "choosesortset" attribute to the same "table_dynamic" element and assign it the unique id of the custom sort set that we created above ("Defense").

Gear and Load Limit (Savage)

All of the gear that characters carry is now in place, so it's an excellent time to re-visit something we didn't complete earlier. The character's load limit is directly influenced by the gear being carried, so we need to hook up that logic and then display it to the user appropriately.

Apply Gear Weights to Encumbrance

Every piece of gear that is carried by a character needs to appropriately apply its weight towards the encumbrance and load limit for that character. Both the encumbrance and load limit are managed via resources that we created earlier, so all we need to do is have every piece of gear subtract its weight from the encumbrance resource. Once that's done, it will automatically be accessed by the load limit resource. The Skeleton files provide an Eval script on the "Gear" component that will handle all of the details for us. All we need to do is modify that script so that it applies the weight to the proper target - the encumbrance resource.

Open the file "equipment.str" and locate the "Gear" component, then find the Eval script that accrues weights into the character. At the bottom of the script, there is a line that is commented out that accrues weight into a dummy resource. Remove the comment and then change the line of code to tally the gear weight into the "resEncumb" resource. All gear on the character should now be properly tracked against both the encumbrance and load limit.

SHOW LOAD LIMIT

We now need to show the encumbrance and load limit somewhere appropriate within HL. Probably the best place to show them would be on the "Basics" tab, beneath the table of attributes. Since we have two pieces of information already and there will likely be additional information that we'll want to display to the user, we'll create a new table portal. We'll also create a new separator beneath the attributes, placing our table beneath the separator and providing a very similar look to how things are handled in the righthand column of the "Basics" tab. All of this will take place within the file "tab_basics.dat".

Before we create a new table portal, we're going to need to identify the specific picks that should be shown within the table. This is most easily accomplished by defining a new tag that is specifically for this purpose. We'll add the tag to the "Helper" tag group and call it "Status", since all the information we'll show in this table is to convey status details about the character to the user. This new tag must be defined in the file "tags.1st".

Our new table portal should look almost the same as the existing "baCreation" table, so we start by copying that table. Then we can tweak the id to something unique, such as "baStatus". We can re-use the exact same template for showing the items in the new table as are used in the existing "baCreation" table, since all it really does is show the name and the "resShort" field of each resource in the table. Lastly, we change the tag expression in the "list" element to limit the items shown to those with the "Helper.Status" tag. This yields a new portal that should look similar to the one below.

```
<portal
  id="baStatus"
  style="tblInvis">
  <table_fixed
    component="Resource"
    template="baResource"
    sortset="explicit"
    scroller="no">
    <list>Helper.Status</list>
    </table_fixed>
  </portal>
```

At this point, the new portal has been defined, but it has not yet been integrated into the layout. So we need to add both the portal and a new separator the layout. Once added, the new portal and template need to be properly sized, rendered, and positioned within the layout. The revisions to the "layout" element should end up looking very similar to those shown below.

```
<layout
  id="basics">
  <portalref portal="baAttrib" taborder="10"/>
  <portalref portal="baStatus" taborder="20"/>
  <portalref portal="baTrait" taborder="30"/>
  <portalref portal="baRank" taborder="40"/>
  <portalref portal="baCreation" taborder="50"/>
  <portalref reference="separator1" portal="Horizontal"/>
  <portalref reference="separator2" portal="Horizontal"/>
  <portalref reference="separator3" portal="Horizontal"/>
  <position><![CDATA[
    ~size and position the attributes table in the top left; we set
the height to
    ~the full layout height, but the table will only use the space it
needs
    portal[baAttrib].width = width / 2 – 5
    portal[baAttrib].left = 0
    portal[baAttrib].height = height
    ~size and position the traits table in the upper right
    portal[baTrait].width = portal[baAttrib].width
    portal[baTrait].left = width - portal[baTrait].width
    portal[baTrait].height = height

    ~set the separator widths
    portal[separator1].width = portal[baTrait].width – 50
    portal[separator2].width = portal[baTrait].width – 50
    portal[separator3].width = portal[baAttrib].width - 50

    ~position the first separator beneath the derived traits
    portal[separator1].top = portal[baTrait].bottom + 15
    portal[separator1].left = portal[baTrait].left +
(portal[baTrait].width -  portal[separator1].width) / 2

    ~size and position the rank details table beneath the first
separator
    portal[baRank].width = portal[baTrait].width
    portal[baRank].left = portal[baTrait].left
    portal[baRank].top = portal[separator1].bottom + 15

    ~position the second separator beneath the rank details
    portal[separator2].top = portal[baRank].bottom + 15
    portal[separator2].left = portal[baRank].left +
(portal[baRank].width -  portal[separator2].width) / 2

    ~size and position the creation details table beneath the
second separator
    portal[baCreation].width = portal[baTrait].width
    portal[baCreation].left = portal[baTrait].left
    portal[baCreation].top = portal[separator2].bottom + 15
    portal[baCreation].height = height - portal[baCreation].top

    ~position the third separator beneath the attributes
    portal[separator3].top = portal[baAttrib].bottom + 20
    portal[separator3].left = portal[baAttrib].left +
(portal[baAttrib].width -  portal[separator3].width) / 2

    ~size and position the status details table beneath the third
separator
    portal[baStatus].width = portal[baAttrib].width
    portal[baStatus].left = portal[baAttrib].left
    portal[baStatus].top = portal[separator3].bottom + 15
    portal[baStatus].height = height - portal[baStatus].top
    ]]></position>
  </layout>
```

The final step in getting the encumbrance and load limit to appear is to locate the resource things and assign them the necessary tags. Open the file "thing_miscellaneous.dat" and you can find the two resources - the unique ids are "resEncumb" and "resLoadLim". We need to assign both things the "Helper.Status" tag so that they will be included in the table. Since the table uses an explicit sort order, we also need to assign one resource an "explicit.1" tag and the other an "explicit.2" tag, thereby specifying the order in which the resources will appear within the table portal.

NON-STANDARD DISPLAY

Both the encumbrance and load limit are showing up to the user now, but the information being shown is not very useful. For the encumbrance, we need to show the current weight being carried and the next load limit threshold, and we need to do it in a way that is intuitively obvious to the user. For the load limit, we should show the current load limit multiplier being utilized and any corresponding penalty that is being applied for exceeding the basic load limit. Ideally, we should also color-highlight the load limit information, putting it in yellow when any penalty is being applied and in red if the character has exceeded the standard limit for what can be carried. Our problem is that the table is hard-wired to show

the "resShort" field for each resource, and there's no way to actually change that behavior.

The trick is that the "resShort" field is synthesized via an Eval script within the "Gear" component. We can override the contents generated by the component if we schedule our own script on the thing that runs after the one on the component. By taking a peek at the file "equipment.str", we can determine the timing of the component Eval script. So we define new Eval scripts on the two resources that we schedule to occur a little bit later in the evaluation process. When the scripts run on the picks, it will clobber whatever contents were generated for the "resShort" field by the component. Our new Eval script for the "resEncumb" resource should look very close to the one below.

```
<eval index="2" phase="Render" priority="10000"><![CDATA[
  ~show the total weight carried and the maximum for the
current load level
  field[resShort].text = field[resSpent].value & " / " &
field[resMax].value
  ]]></eval>
```

For the "resLoadLim" resource, the new Eval script entails a bit more logic, but it should end up looking similar to the one below.

```
<eval index="3" phase="Render" priority="10000"><![CDATA[
  ~start with our load limit multiple
  var short as string
  short = field[resMax].value

  ~calculate how many multiples we've consumed
  var multiples as number
  multiples = 1
  if (field[resExtra].value >= field[resMax].value) then
    multiples += round(field[resExtra].value /
field[resMax].value,0,-1)
    endif

  ~append the number of multiples carried
  short &= "x" & multiples

  ~append any adjustment incurred for overage
  if (multiples > 1) then
    var overage as number
    overage = multiples – 1
    short &= " (-" & overage & ")"
    endif

  ~color highlight based on the severity of our overage
  if (multiples > 4) then
    short = "{text ff0000}" & short
  elseif (multiples > 1) then
    short = "{text ffff00}" & short
    endif

  ~put the result into the field
  field[resShort].text = short
  ]]></eval>
```

Natural Weapons (Savage)

When we added races earlier, we made a note to add their natural weapons after everything was in place for weapons. We've reached that point, so we might as well get the natural weapons working now.

General Approach

Many game systems have races that confer special nuances to the unarmed attacks of its members. Consequently, the "Unarmed Strike" weapon (unique id "wpUnarmed") has been designed so that it can be readily overridden for situations like this. Fields like the weapon damage can be changed via an Eval script, and new tags for special abilities can be easily assigned as well.

For each race with a natural weapon, an appropriate ability is already defined and bootstrapped by the race. This ability can have an Eval script defined for it that properly overrides the "Unarmed Attack" weapon with the appropriate behaviors for the race. A separate Eval script on the ability will manage any other special behaviors associated with the natural weapon. This approach works well for situations like the Rakashan race, where their claws serve both as a natural weapon and confer a bonus when climbing.

Defining the Natural Weapons

The Rakashans possess a natural weapon in the form of "Claws", and they do "Str+d6" damage. So we'll start by adding a new Eval script to the Rakashan Claws ability (unique id "abClawsRk"), which can be found in the file "thing_races.dat". The only thing we need to do for this ability is change the name and damage of the "Unarmed Strike" weapon to something more appropriate. Changing the damage requires that we forward the appropriate "WeaponDie" tag before it is referenced. We'll use the "focus" mechanism so that we only need to identify the "Unarmed Strike" weapon a single time and can then operate on it easily. This yields a new Eval script that looks similar to the one below.

```
<eval index="2" phase="Initialize" priority="1000"><![CDATA[
  perform hero.child[wpUnarmed].setfocus
  focus.field[livename].text = "Claws"
  perform focus.assign[WeaponDie.3]
  ]]></eval>
```

At this point, anytime the Rakashan race is selected, the built-in "Unarmed Strike" attack will be customized so that it appears to be a completely different attack for the character.

The Saurian race is very similar to the Rakashans. All that is needed is to revise the name and damage of the "Unarmed Strike" to yield an appropriate replacement. If the natural weapon confers special abilities that require new weapon tags, those can be assigned to the "Unarmed Strike" weapon in the same way that fields are being overridden.

Special Weapons (Savage)

Now that all the basic gear is in place and being handled properly, we can go back and start adding special kinds of gear. We'll start with the various "Special Weapons" given in the Savage Worlds rulebook.

New Component and Component Set

Special weapons are virtually the same as ranged weapons in terms of what is needed to manage them properly. However, it's critical that we clearly delineate between ranged weapons and special weapons. Consequently, we need to define a new component and component set for special weapons. Provided that we ensure that we have a clear and simple means of distinguishing between ranged and special weapons, we are free to re-use all the logic for ranged weapons. This allows us to define a component for special weapons that does nothing and re-use the "WeapRange" component in our

new component set. We can rely on the automatically defined component tag to identify a special weapon distinctly from a ranged weapon. Based on this, our new component and component set should look similar to the following.

```
<component
  id="WeapSpec"
  name="Special Weapon"
  autocompset="no">
  </component>

<compset
  id="SpecWeap"
  stackable="yes">
  <compref component="BaseWeapon"/>
  <compref component="WeapRange"/>
  <compref component="WeapSpec"/>
  <compref component="Equippable"/>
  <compref component="Gear"/>    </compset>
```

Distinguish from Ranged Weapons

All special weapons will identify themselves as ranged weapons now because both possess the "component.WeapRange" tag. Anywhere that we currently rely on this tag to identify a ranged weapon needs to be revised to properly exclude special weapons. In order to determine whether a given weapon is solely a ranged weapon, we need to verify that it possesses the "component.WeapRange" tag and that it does not possess the "component.WeapSpec" tag.

Scanning through our data files, most references to the component tag are simply to determine whether to show range information for the weapon, so we need to preserve those references intact. The only case where we need to actively apply this new restriction is within the "arRange" table portal on the "Armory" tab, which will be found in the file "tab_armory.dat". The List tag expression for the table must be revised to explicitly exclude things that possess the "component.WeapSpec" tag. The new version of this element is shown below.

```
<list>component.WeapRange &  !component.WeapSpec</list>
```

New Behaviors

There are a variety of new behaviors the must be handled in support of special weapons. The first new behavior is that some special weapons possess a standard range, while others (e.g. mines) have no range. For the latter group of special weapons, we need to appropriately deal with the lack of range by displaying something appropriate. The "WeapRange" component already possesses a "wpRange" field that uses a Finalize script to synthesize the range appropriately. So we'll modify this Finalize script to handle this new case, resulting in a revised script that looks similar to the following.

```
 if (tagis[Weapon.SpecRange] <> 0) then
    @text = "Special"
  elseif (field[wpShort].value + field[wpMedium].value +
field[wpLong].value = 0) then
    @text = "--"
  else
    @text = field[wpShort].value & "/" & field[wpMedium].value &
"/" & field[wpLong].value
    endif
```

In order to sort special weapons by grouping in the same way that we do for other weapons, we'll need to define a number of new "WeaponType" tags in the file "tags.1st". Each new tag will correspond to a grouping that is used within the Savage Worlds rulebook, and we'll also preserve the order used within the rulebook. The new tags should consist of the following:

```
  <value id="SpecCannon" name="Special: Cannons"
order="20"/>
  <value id="SpecRocket" name="Special: Rocket Launchers"
order="21"/>
  <value id="SpecMines" name="Special: Mines" order="22"/>
  <value id="SpecFlame" name="Special: Flamethrowers"
order="23"/>
  <value id="SpecGren" name="Special: Grenades" order="24"/>
  <value id="SpecExplos" name="Special: Explosives"
order="25"/>
```

The final new behavior that we must handle is an assortment of new special abilities. The new abilities are those pertaining to the various templates that are used by the different weapons. By adding new "Weapon" tags for these abilities, simply assigning the tags to the weapons will automatically incorporate the details into the description and other output for each weapon, thanks to the revisions we made earlier. The new tags should consist of the following:

```
<value id="MedBurst" name="Medium Burst Template"/>
 <value id="SmallBurst" name="Small Burst Template"/>
 <value id="LargeBurst" name="Large Burst Template"/>
 <value id="ConeTempl" name="Cone Template"/>
```

Table for Special Weapons

All of the mechanics are now in place for special weapons, but we still need to make them accessible to the user. We'll add a new table portal to the "Armory" tab for this purpose. Since all of the behaviors will be very similar to those for the existing ranged weapons table, we'll start by copying the "arRange" portal. We can then rename it to "arSpecial" and change it to only list things belonging to the "WeapSpec" component. We need to revise the List tag expression to only process the appropriate weapons, then we can change the various strings used in scripts to properly indicate that we're manipulating special weapons instead of ranged weapons. Everything else can remain the same. Putting it all together yields the revised table portal shown below.

```
 <portal
   id="arSpecial"    style="tblNormal">
   <table_dynamic
     component="WeapSpec"
     showtemplate="arWpnPick"
     choosetemplate="arWpnThing"
     choosesortset="Weapon"
     buytemplate="BuyCash"
     selltemplate="SellCash"
     descwidth="275">
     <list>component.WeapSpec</list>
     <candidate inheritlist="yes">!Equipment.Natural</candidate>
     <titlebar><![CDATA[
       @text = "Select Special Weapons to Purchase from the List
Below"
       ]]></titlebar>
     <headertitle><![CDATA[
       @text = "Special Weapons"
       ]]></headertitle>
     <additem><![CDATA[
       @text = "Add New Special Weapons"
       ]]></additem>
```

```
    </table_dynamic>
  </portal>
```

Positioning Everything

We've got a new table portal defined now, so we need to integrate it into the layout that controls what's shown for the tab. This entails adding an appropriate "portalref" element to the layout and then properly sizing and positioning the new portal. We'll put the table at the bottom of the other portals, since the others are more likely to see regular access by users. We'll reserve space to show at least two items in the table, unless there are fewer items selected by the user. At the end, if there is still unused space left, we'll extend the table downward to use up whatever space is available. This makes it possible to preserve all of the existing logic for adaptively sizing the other table portals within the layout. All we need to do is integrate our new portal in appropriately. The revised layout element that cleanly integrates the new portal is presented below.

```
<layout
  id="armory">
  <portalref portal="arMelee" taborder="10"/>
  <portalref portal="arRange" taborder="20"/>
  <portalref portal="arDefense" taborder="30"/>
  <portalref portal="arSpecial" taborder="40"/>
  <position><![CDATA[
   ~determine the gap to use between tables
   var gap as number
   gap = 10

  ~set the width of all tables to the full width of the layout
   portal[arMelee].width = width
   portal[arRange].width = width
   portal[arDefense].width = width
   portal[arSpecial].width = width

  ~position the special weapons table at the bottom, allowing for
at most two rows
   portal[arSpecial].maxrows = 2
   portal[arSpecial].top = height - portal[arSpecial].height

  ~determine the height remaining that can be used by other
tables
   var ht as number
   ht = height - portal[arSpecial].height - gap

  ~position the armor/shield table above special weapons,
allowing for at most two rows
   portal[arDefense].maxrows = 2
   portal[arDefense].top = ht - portal[arDefense].height

  ~position the melee table at the top
   portal[arMelee].top = 0

  ~set the heights of the two weapon tables to use all the space
available
   portal[arMelee].height = ht
   portal[arRange].height = ht

  ~determine how much space we have left for the two tables;
be sure to exclude
   ~the extra title and the extra spacing we'll use inbetween
   ~NOTE! If a value of 10 is added to the bottom coordinate of
a portal, the
   ~net value will yield an actual GAP of one less. For example,
if the bottom
   ~is at pixel 15, that pixel is part of the physical height of the
portal. If
   ~you add 10 to that position for the next portal, it starts on
pixel 25, so
   ~pixel 25 is part of the next portal. That means that pixels 16
   ~represent the dead space inbetween, which is a span of 9
pixels. We have to
   ~factor this detail in when adjusting the space remaining by
our gaps.
   var remain as number
   remain = portal[arDefense].top - portal[arMelee].top
   remain -= (gap - 1) * 2

  ~if the height of both tables exceeds what we have left, we
need to divvy up
   ~that space between the two tables
   if (portal[arMelee].height + portal[arRange].height > remain)
then

    ~if the melee table is less than half the space, limit the
ranged table
     ~to whatever space is leftover
     if (portal[arMelee].height < remain / 2) then
       portal[arRange].height = remain - portal[arMelee].height

    ~if the ranged table is less than half the space, limit the
melee table
     ~to whatever space is leftover
     elseif (portal[arRange].height < remain / 2) then
       portal[arMelee].height = remain - portal[arRange].height

    ~otherwise, both tables are larger than half the space, so we
need to limit
     ~the height of both of them
     ~NOTE! If we just divide the remaining amount by two and
set both tables to
     ~that height, we could end up with both tables being
truncated by more than
     ~a half item, with the combined height being a full item short
of taking up
     ~the full space. So we have to set the height of one table to
half the
     ~remaining space, then subtract that table's final height from
our remaining
     ~space, and finally set that as the height for the second
table
.      else
       portal[arRange].height = remain / 2
       portal[arMelee].height = remain - portal[arRange].height
       endif
     endif

  ~position the ranged weapons table beneath the melee table
   portal[arRange].top = portal[arMelee].bottom + gap

  ~position the armor/shields table beneath the ranged
weapons table
   ~NOTE! we already positioned this table, but the above logic
could result in
   ~a gap between the tables, so we close that gap by
repositioning again
   portal[arDefense].top = portal[arRange].bottom + gap

   ~set the height of the armor/shields table to the whatever
height is left;
   ~if the armor list is long and the weapon lists are short, this
will show as
   ~much armor as there is remaining room to accommodate
   portal[arDefense].height = ht - portal[arDefense].top

   ~position the special weapons table beneath the
armor/shields table
   portal[arSpecial].top = portal[arDefense].bottom + gap

   ~set the height of the special weapons table to the whatever
height is left;
   ~if the above tables are short, this will show as many special
weapons as
   ~there is remaining room to accommodate
```

```
    portal[arSpecial].height = height - portal[arSpecial].top
    ]]></position>
    </layout>
```

EXPANDING OUR COVERAGE

CONFERRING EDGES AND HINDRANCES (SAVAGE)

There are a handful of interesting mechanics in Savage Worlds that all hinge on a similar capability within the data files. Edges and hindrances can be automatically conferred by another selection, such as races, injuries, fright, etc.

DETECTING BOOTSTRAPPED PICKS

When an edge or hindrance is conferred by another selection, it must be bootstrapped by that selection. Any edge or hindrance that is handled this way needs to be treated specially, since it should not count as having cost an advance (for edges) nor should it count as earning rewards (for hindrances). This entails special handling that must be instrumented at the component level so that it works smoothly for all instances. Since both edges and hindrances accrue advances and rewards separately, we need to implement the same basic logic within both the "Edge" and "Hindrance" components.

We can determine whether a pick has been bootstrapped by another pick via use of the "isroot" target reference. If "isroot" returns non-zero, the pick has a root pick that bootstrapped it into existence. If we know a pick has been bootstrapped, then we can avoid accruing the appropriate cost into the resource that tracks the edges or hindrances. We can modify the existing Eval scripts that accrue the cost of edges and hindrances by adding this new logic.

WHAT IF BOOTSTRAPPED AND USER-ADDED?

Unfortunately, this isn't a complete solution for what we need. The problem is that users may have already added the edge or hindrance manually, prior to when it is bootstrapped. This results in a conflict. The question becomes whether we should just ignore the user-added selection or treat it normally. So let's consider a couple of examples to figure out how to handle this. The first example is a user who adds an edge to a character during creation and then chooses a custom race that confers the same edge. In this instance, we should probably not count the edge against the character and should warn the user so that he can delete the edge that was manually added. Similarly, consider a user who adds a hindrance to a character during creation and then selects a race that confers the same hindrance. We should probably not count the hindrance and should warn the user about the conflict. Unfortunately, there is a flipside to this. Consider a user who selects the "Ugly" hindrance during character creation and then later incurs a permanent injury that confers the "Ugly" hindrance. In this situation, we simply want to ignore the bootstrapping of "Ugly", since the character is already ugly and is essentially no further impacted by the injury.

These various examples are in direct conflict with one another, and we can't do it both ways. This means that we have to choose one method and simply live with it, even though there will be times that it's not the optimal solution. When considering the two alternatives, we need to consider which ones are (a) more likely to occur and (b) more likely to yield results that are confusing to the user. The examples where we should not count the edge or hindrance only occur during character creation. Furthermore, these examples are unlikely to occur, since races rarely confer edges and hindrances, and the examples rely upon the user manually selecting an edge or hindrance before selecting the character's race. In contrast, situations where the character incurs a permanent injury that replicates a hindrance chosen at creation is much more likely to occur. And if we stop accruing a hindrance that was taken at creation, the character will begin reporting validation errors that don't actually exist. So our choice is pretty clear: we need to always accrue the cost of edges and hindrances that are user-added, even if they are also bootstrapped by other selections.

On the surface, this conclusion might seem to imply that we don't actually need to change anything in the code. However, there is a critical detail in our conclusion that does necessitate a change. In each of our examples, we examined what to do when the user adds an edge/hindrance and that edge/hindrance is bootstrapped by some other selection. However, we also need to deal with edges and hindrances that are only bootstrapped by another selection. In this circumstance, we do not want to accrue the cost of the edge/hindrance.

USER-ADDED ONLY

This means that we only want to accrue the cost if the edge/hindrance was added by the user. If it is also bootstrapped, we still treat it as having been added by the user and accrue it normally. However, if it is only added via bootstrapping, we need to skip it. Fortunately, there is the "isuser" target reference that will tell us whether a given pick has been added by the user. So we'll use this to ascertain whether to accrue the cost for each edge and hindrance. We can now modify the existing Eval scripts to implement the appropriate logic. The revised script for the "Edge" component should end up looking similar to the following.

```
 <eval value="2" phase="Setup" priority="5000"><![CDATA[
   ~if this edge is not added directly to the hero (i.e. an advance),
skip it entirely
   if (origin.ishero = 0) then
     done
     endif

   ~if this edge was not added by the user, skip it entirely
   if (isuser = 0) then
     done
     endif

   ~consume another edge slot
   #resspent[resEdge] += 1
   ]]></eval>
```

Applying the equivalent change to the Eval script for the "Hindrance" component yields the revised script below.

```
 <eval value="2" phase="Setup" priority="5000"><![CDATA[
   ~if this hindrance was not add by the user, skip it entirely
   if (isuser = 0) then
     done
     endif

   ~consume another hindrance slot or two, depending on the
severity
   #resmax[resHinder] += field[hinMajor].value + 1
   ]]></eval>
```

ALL THUMBS AGAIN

At this point, we need to revisit something that we did early in our development. When we added the Elven race, we opted to add a new ability that paralleled the "All Thumbs" hindrance so that we could avoid having to deal with hindrances being bootstrapped by races. Well, we just dealt with that situation, so we no longer need to have an ability to mimic the hindrance. We can now go back into the file "thing_races.dat" and do two things. First, we can modify the "Elven" race to bootstrap the "All Thumbs" hindrance instead of the ability. This entails changing the unique id being bootstrapped from "abAllThumb" to "hinThumbs". Second, we can delete the "All Thumbs" ability that is no longer needed nor used.

INJURIES (SAVAGE)

Savage Worlds includes the notion of injuries that can be attributed to characters, with corresponding penalties being applied.

PERMANENT VERSUS TEMPORARY

There are both temporary and permanent injuries to worry about within Savage Worlds. When a temporary injury is incurred, it can simply be managed on an interim basis by applying a temporary adjustment to the character on the In-Play tab. However, a permanent injury is an entirely different situation, as it has subsequent implications on character advancement. This is because some injuries will confer penalties to attributes, which impacts advancements to skills that are linked to those attributes (i.e. skills may transition from being less than the linked attribute to equal).

The upshot of this is that permanent injuries need to be treated as advancements. That way, any attribute adjustments due to the injuries will properly impact subsequent advancements to skills. There are a number of steps that need to be taken in order to manage permanent injuries via advancements. The following paragraphs will outline each of them.

INJURY COMPONENT

The first step is to create a new component and component set for handling injuries. We'll call both of them "Injury". In order to incorporate the appropriate handling for advancements, the component set needs to include the "CanAdvance" component. We don't need any special fields or behaviors for injuries, since all of the mechanisms we need are provided by the "CanAdvance" component. However, we define a new component just to be safe, since this will make it easy for us to add new behaviors in the future, if necessary. This yields the following definitions for both the component and component set.

```
<component
  id="Injury"
  name="Injury"
  autocompset="no">
  </component>

<compset    id="Injury">
  <compref component="Injury"/>
  <compref component="CanAdvance"/>
  </compset>
```

BOOTSTRAPPING HINDRANCES

We can now assess exactly what the behaviors should be for each of the different injuries. A couple of the injuries will consist of nothing more than description text. A number of others need to apply adjustments to attributes and/or other traits (e.g. Pace). And the final ones will automatically confer a hindrance upon the character.

Due to the need to confer hindrances via injuries, we must be sure to modify the "Hindrance" component set to also inherit the "CanAdvance" component. This will ensure that hindrances can be bootstrapped by injuries and properly displaced up to the hero. This results in the revised "Hindrance" component set below.

```
<compset
  id="Hindrance"
  forceunique="yes">
  <compref component="Hindrance"/>
  <compref component="Ability"/>
  <compref component="Domain"/>
  <compref component="SpecialTab"/>
  <compref component="CanAdvance"/>
  </compset>
```

ADD INJURIES

The next step is the creation of the various injury things. In order to keep them all together, we'll create a new data file exclusively for injuries. We'll name the file "thing_injuries.dat" and we can start with any of the current data files, then we can strip out everything but the outer framework so that we have an initial file that we can put to use.

An example of an injury that applies an attribute adjustment is the "Broken Guts" injury, which degrades the character's Agility by one die type. When we apply adjustments to attributes and other traits, we'll modify the "trtBonus" field on the trait. This will keep injury adjustments distinct from normal in-play adjustments. The sample thing below reflects a suitable definition for the "Broken Guts" injury.

```
<thing
  id="injBroken"
  name="Broken Guts"
  compset="Injury"
  description="Description goes here">
  <eval index="1" phase="PreTraits" priority="5000">
    <before name="Calc trtFinal"/><![CDATA[
    #traitbonus[attrAgi] -= 1
    ]]></eval>
  </thing>
```

The other interesting type of injury is one that bootstraps an appropriate hindrance. For example, the "Hideous Scar" injury confers the "Ugly" hindrance to the character. When we bootstrap the hindrance, it is critical that we also ensure that the "Helper.Displace" tag is assigned to the hindrance. Doing so guarantees that the hindrance will be properly displaced onto the character (from the advancement gizmo), which allows it to be clearly visible to the user (e.g. shown on the "Edges" tab). This yields a thing that should look like the following.

```
<thing
  id="injScar"
  name="Hideous Scar"
  compset="Injury"
  isunique="yes"
  description="Description goes here">
  <bootstrap thing="hinUgly">
    <autotag group="Helper" tag="Displace"/>
    </bootstrap>
```

```
</thing>
```

Note that one of the above examples is designated as "unique" and the other is not. Appropriate injuries must be designated as unique to avoid duplicate selection by the user. There are some that have an effect which makes it pointless to select the injury more than once (e.g. "Unmentionables"). These injuries should be marked as unique so that they are not able to be selected multiple times. Other injuries that can occur multiple times can omit this designation. You can now either define the remaining injuries yourself or pull them from the completed Savage Worlds data files.

INJURY ADVANCEMENT

Once the injuries are in place, we can setup the handling for the new advancement thing that will be used to add and select permanent injuries. Our advancement can now be configured to force the user to select a new injury. We set the "cost" of the advancement to zero so that adding a permanent injury does not consume a normal advancement slot. The net result is a thing that looks very similar to the following.

```
<thing
  id="advInjury"
  name="Permanent Injury"
  compset="Advance"
  description="Select this advance to apply a permanent injury
to the character. The effects of temporary injuries should be
applied via temporary adjustments on the In-Play tab.">
  <fieldval field="advAction" value="Permanent Injury"/>
  <fieldval field="advDynamic" value="component.Injury"/>
  <fieldval field="advCost" value="0"/>
  <tag group="Advance" tag="AddNew"/>
  <child entity="Advance">
    <tag group="Advance" tag="MustChoose"/>
  </child>
</thing>
```

REVISE INTERFACE

The one big drawback with this mechanism is that injuries are added just like new skills. This results in the interface presenting injuries for selection as if the user is selecting a new skill. The mechanism looks extremely odd. The solution is to introduce a new type of advancement and adapt the mechanism to work well for injuries as well. For the new advancement type, we'll define a new "Advance.Injury" tag within the file "advancements.core". This way, that we can readily distinguish injury advancements from other advancements.

We can now open the file "form_advance.dat" and look into adapting the current advancement mechanism for better use with injuries. Ideally, we could adapt the "advNew" chooser portal for use with injury selection. Unfortunately, the chooser lacks sufficient contextual details to be able to adapt the current portal. This means that we need to duplicate the chooser portal, rename it, and then look to adapt the copy. For simplicity, we'll start by duplicating the "advNew" chooser portal. After revising the chooser for suitability to injuries, we should have a new portal that looks very similar to the following.

```
<portal
  id="advInjury"
  style="chsNormal"    width="275">
  <chooser_table
    component="none"
    prereqtarget="hero"
    candidatepick="advDetails"
    candidatefield="advTagexpr"
    descwidth="350">
    <denytag usehero="yes" container="IsAdvance"
thing="IsAdvance"/>
    <autotag group="Helper" tag="Displace"/>
    <autotag group="Advance" tag="Gizmo"/>
    <chosen><![CDATA[
      if (@ispick = 0) then
        @text = "{text ff0000}Select New Injury"
      else
        @text = "New Injury: " & field[name].text
        endif
      ]]></chosen>
    <titlebar><![CDATA[
      @text = "Choose the New Injury to Add"
      ]]></titlebar>
    <description/>
    </chooser_table>
  </portal>
```

The final thing we need to do is add the new portal to the layout. Once added, we can properly position it within the layout and control its visibility. This can be seen in the revised layout shown below.

```
<layout
  id="advance">
  <portalref portal="advNew" taborder="20"/>
  <portalref portal="advInjury" taborder="20"/>
  <portalref portal="advBoost" taborder="20"/>
  <templateref template="advTitle" thing="advDetails"
taborder="10"/>
  <templateref template="advStatic" thing="advDetails"
taborder="20"/>
  <templateref template="advDomain" thing="advDetails"
taborder="30"/>
  <templateref template="advNotes" thing="advDetails"
taborder="40"/>
  <position><![CDATA[
    ~setup a width that is generally reasonable
    width = 475

    ~set the fixed dimensions and render the title template
    template[advTitle].width = width
    perform template[advTitle].render

    ~position the chooser to add a new advancement
    portal[advNew].left = 20
    portal[advNew].top = template[advTitle].bottom + 30
    portal[advNew].width = 310

    ~position the chooser for injuries in the same place
    portal[advInjury].left = portal[advNew].left
    portal[advInjury].top = portal[advNew].top
    portal[advInjury].width = portal[advNew].width

    ~position the chooser for boosting in the same place
    portal[advBoost].left = portal[advNew].left
    portal[advBoost].top = portal[advNew].top
    portal[advBoost].width = portal[advNew].width

    ~position the static template in the same place (for non-
editable items)
    template[advStatic].left = portal[advNew].left
    template[advStatic].top = portal[advNew].top
    template[advStatic].width = portal[advNew].width

    ~hide all of the different portals/templates - we'll show one
below
    template[advStatic].visible = 0
    portal[advNew].visible = 0
```

```
      portal[advInjury].visible = 0
      portal[advBoost].visible = 0

    ~if this is NOT the newest advancement, show the static
template so the user
    ~can't change anything; otherwise, determine which chooser
should be shown
    if (container.parent.tagis[Advance.Newest] = 0) then
      template[advStatic].visible = 1
    elseif (container.parent.tagis[Advance.AddNew] <> 0) then
      portal[advNew].visible = 1
    elseif (container.parent.tagis[Advance.Injury] <> 0) then
      portal[advInjury].visible = 1
    else
      portal[advBoost].visible = 1
      endif

    ~position the domain template
    template[advDomain].left = portal[advNew].left + 15
    template[advDomain].top = portal[advNew].bottom + 8

    ~position the notes template
    template[advNotes].left = portal[advNew].left

    ~setup fixed dimensions of our templates
    template[advDomain].width = portal[advNew].width -
template[advDomain].left
    template[advNotes].width = width - template[advNotes].left *
2

    ~render all remaining templates to finalize their dimensions
    perform template[advDomain].render
    perform template[advNotes].render

    ~determine whether the domain template should be shown
    ~Note: The final test below ensures that we only let the user
change the domain
    ~ when editing the newest advancement and not an old one.
    var isdomain as number
    isdomain = container.parent.tagis[Advance.AddNew]
    if (isdomain <> 0) then
      isdomain =
container.firstchild["Advance.Gizmo"].tagis[User.NeedDomain]
      if (isdomain <> 0) then
        isdomain = container.parent.tagis[Advance.Newest]
        endif
      endif
    template[advDomain].visible = isdomain

    ~position the notes at the bottom
    if (isdomain <> 0) then
      template[advNotes].top = template[advDomain].bottom + 20
    else
      template[advNotes].top = portal[advNew].bottom + 20
      endif

    ~determine our overall height
    height = template[advNotes].bottom
    ]]></position>
  </layout>
```

FRIGHT (SAVAGE)

Fright is a Savage Worlds mechanic that works very similarly to injuries. If a character suffers a severe enough scare, they can become subject to a particular fright behavior on a permanent basis. Some frights confer a hindrance in the same way that injuries do, and we need to handle that properly.

FRIGHTS AS ADVANCEMENT

When a character has a severe scare and incurs a fright, that becomes a permanent effect on the character. Since frights do not apply adjustments to attributes, there is no requirement for frights to be handled as advancements. However, the basic logic for treating them as advancements is already in place, and it should also make sense to users. So we'd be well-served by setting up frights as a new type of advancement.

Before we launch into doing that, though, we should first consider the bigger picture. Frights either confer temporary penalties or a hindrance of some sort. It's also possible through the course of play that a character will become subject to a new hindrance that has no association with a fright. Consequently, it's probably a better approach for us to make it possible to assign a new hindrance to a character as a advancement. Any hindrance conferred due to fright represents a special case, while adding hindrances in general solves the general case.

HINDRANCES AS ADVANCEMENT

We can add hindrances via their own new advancement very easily. The only special detail that we have to make sure of is that we must ensure the cost of a hindrance advance is zero, just like we did for injuries. Beyond that, adding hindrances via advancement works just like adding a new skill or edge. The corresponding advancement thing should look a lot like the example provided below.

```
  <thing
    id="advHinder"
    name="New Hindrance"
    compset="Advance"
    description="Select this advance to ascribe a new hindrance
to the character during play. When acquired as an advancement,
no offsetting rewards are gained for the hindrance.">
    <fieldval field="advAction" value="New Hindrance"/>
    <fieldval field="advDynamic" value="component.Hindrance"/>
    <fieldval field="advCost" value="0"/>
    <tag group="Advance" tag="AddNew"/>
    <child entity="Advance">
      <tag group="Advance" tag="MustChoose"/>
      </child>
    </thing>
```

ADD FRIGHTS

There is one fright that confers an effect which is not a standard hindrance - the "Mark of Fear". Consequently, we need to define that effect as its own new hindrance so that it can be selected by the user if it comes up during play. For simplicity, we'll add the new hindrance along with all the other hindrances in the file "thing_hindrances.dat". Our new hindrance should look similar to the one shown below.

```
  <thing
    id="hinMkFear"
    name="Mark of Fear"
    compset="Hindrance"
    isunique="yes"
    description="Description goes here">
    <fieldval field="hinMajor" value="0"/>
    <eval index="1" phase="PreTraits" priority="5000">
      <before name="Calc trtFinal"/><![CDATA[
      #traitbonus[trCharisma] -= 1
      ]]></eval>
    </thing>
```

HINDRANCE ADVANCES DON'T CONFER REWARDS

We have one last detail that needs to be sorted out. Hindrances that are added via advances need to not confer reward points to the

character. Looking at the current Eval script for the "Hindrance" component, rewards are accrued for any hindrance that is user-added. Unfortunately, any hindrance added as an advancement is user-added, so it will confer reward points. So we need to revise the script in some way to preclude advancements from being accrued. One detail that we can always rely upon is that every advance always possesses an "Advance.?" tag. This means that we can simply check for an advancement tag and ignore the accrual if we find such a tag. A revised version of the Eval script that incorporates this change is presented below.

```
<eval index="3" phase="Setup" priority="5000"><![CDATA[
  ~if this hindrance was not add by the user, skip it entirely
  if (isuser = 0) then
    done
    endif

  ~if this hindrace was added as an advance, skip it entirely
  if (tagis[Advance.?] <> 0) then
    done
    endif

  ~consume another hindrance slot or two, depending on the
severity
  #resmax[resHinder] += field[hinMajor].value + 1
  ]]></eval>
```

VEHICLES

BASIC VEHICLES (SAVAGE)

There are two varieties of vehicles defined within the Savage Worlds rulebook. Basic vehicles have all the standard characteristics shared by all vehicles, such as civilian cars. Complex vehicles are outfitted with a variety of weapons that must be handled appropriately and introduce a number of additional wrinkles. So we'll focus on the mechanics associated with basic vehicle in this section.

NEW TAGS

Vehicles behave similarly to weapons and armor within Savage Worlds. As such, we'll be using the same basic approach in implementing them. The first thing we'll need to address is the assortment of tags that will be needed for vehicles. After taking a moment to review how vehicles are handled, there are three different facets of vehicles that we need to keep distinct: the type, the era, and any special characteristics. So we'll create a separate tag group for each of these facets.

We'll start with the vehicle type, of which there are three basic types defined in the core rulebook. These are ground vehicles, aircraft, and boats. Since all of our tags are defined in the file "tags.1st", open that file and locate a suitable spot to insert the new tag group (e.g. after the armor-related tags). We'll make tag group dynamic so that supplements can define new vehicle types, if necessary. We'll use an explicit order, but we'll leave gaps in the ordering so that supplements can insert new vehicle types into the list wherever they want. This yields a tag group definition that looks like the one shown below.

```
<group
  id="VehType"
  dynamic="yes"
  sequence="explicit">
  <value id="Ground" name="Ground Vehicles" order="10"/>
  <value id="Aircraft" order="20"/>
  </group>
```

Next up is the era of the vehicle, and there are four eras addressed in the core rulebook. These are civilian, WWII military, modern military, and futuristic military. We use the same principles as above, allowing for extensibility, and end up with a tag group similar to the one below.

```
<group
  id="VehEra"
  dynamic="yes"
  sequence="explicit">
  <value id="Civilian" order="10"/>
  <value id="WWII" name="WWII Military" order="20"/>
  <value id="Modern" name="Modern Military" order="30"/>
  <value id="Future" name="Futuristic Military" order="40"/>
  </group>
```

The final tag group we need to define is all of the various special characteristics that apply to vehicles. We'll handle this exactly like we do for weapons and armor, which yields a tag group that looks like the one below.

```
<group
  id="Vehicle">
  <value id="Amphib" name="Amphibious"/>
  <value id="Spacecraft"/>
  <value id="Atmosphere" name="Atmospheric"/>
  <value id="Tracked"/>
  <value id="4WD" name="Four-Wheel Drive"/>
  <value id="HvyArmor" name="Heavy Armor"/>
  <value id="Sloped" name="Sloped Armor"/>
  <value id="FixedGun" name="Fixed Gun"/>
  <value id="HvyWeapon" name="Heavy Weapon"/>
  <value id="AirBags" name="Air Bags"/>
  <value id="Stealth" name="Stealh Paint"/>
  <value id="AST" name="Advanced Stealth Tech"/>
  <value id="AMCM" name="Anti-Missile Counter Measures"/>
  <value id="Stabilizer" name="Stabilizer"/>
  <value id="ImpStabil" name="Improved Stabilizer"/>
  <value id="NightVis" name="Night Vision"/>
  <value id="Infrared" name="Infrared Night Vision"/>
  </group>
```

NEW SORT SET

Since we have a variety of factors for organizing vehicles, we're going to need a new sort set to put them in the proper sequence for display. We'll use a sequence that parallels the organization in the rulebook, so we'll sort first by the vehicle type, then the era, and finally by name. This yields a sort set similar to the one below, which can be defined with other sort sets in the file "control.1st".

```
<sortset
  id="Vehicle"
  name="Vehicles By Type, Era, and Name">
  <sortkey isfield="no" id="VehType"/>
  <sortkey isfield="no" id="VehEra"/>
  <sortkey isfield="no" id="_Name_"/>
  </sortset>
```

NEW FIELDS FOR VEHICLES

Vehicles in Savage Worlds have a bunch of fields that need to handled properly. In addition to the fields used in the rulebook, we'll utilize the same mechanism we used for weapons and armor, in which we have a "notes" field where we synthesize all the details for

display. Since we're going to be adding vehicles to the "Gear" tab, we can also associated the component with the appropriate panel. This yields an expanded definition for the "Vehicle" component that is defined in the file "equipment.str". The new component should look similar to below, including a suitable script for synthesizing the notes.

```
<component
  id="Vehicle"
  name="Vehicle"
  panellink="gear"
  autocompset="no">

 <field
   id="vhAccel"
   name="Acceleration"
   type="static"
   maxlength="10">
   </field>

 <field
   id="vhTopSpeed"
   name="Top Speed"
   type="static"
   maxlength="15">
   </field>

 <field
   id="vhClimb"
   name="Climb"
   type="static"
   maxlength="15">
   </field>

 <field
   id="vhTough"
   name="Toughness"
   type="static"
   maxlength="10">
   </field>

 <field
   id="vhArmor"
   name="Armor"
   type="static"
   maxlength="10">
   </field>

 <field
   id="vhCrew"
   name="Crew"
   type="static"
   maxlength="10">
   </field>

 <field
   id="vhSpecial"
   name="Special"
   type="derived"
   maxlength="200">
   </field>

 <field
   id="vhNotes"
   name="Notes"
   type="derived"
   maxlength="250">
   </field>

 <eval index="1" phase="Render" priority="1000"><![CDATA[
   var special as string

  ~append any special attributes appropriately (if any)
  var attribs as string
  if (empty(attribs) = 0) then
    if (empty(special) = 0) then
      special &= ", "
      endif
    special &= attribs
    endif

  ~append any special details for this vehicle
  if (field[vhSpecial].isempty = 0) then
    if (empty(special) = 0) then
      special &= ", "
      endif
    special &= field[vhSpecial].text
    endif

  ~we've synthesized the notes for the vehicle
   field[vhNotes].text = special
   ]]></eval>

</component>
```

VEHICLE COST AS A RANGE

There is a key difference with how vehicles behave from normal gear, though. Many vehicles are given a price range instead of an absolute price. If we want to support this, we need to figure out a way to integrate this different behavior into the way everything is handled for gear. In order to ensure that buy/sell transaction handling all works correctly, we need to retain the use of the "grCost" field and that field must specify an explicit value to be used as the default cost.

Probably the easiest way to handle this is to define a new field in which to display the vehicle cost range. Then we simply need to decide what value we'll use for the explicit "grCost" field. The obvious choices are to always use the lowest price in the range, the highest price in the range, or the mid-point of the range. We'll standardize on the lowest price in the range.

We now need to define the new field in which the cost range will be be given. This can be done with something simple like the field definition below.

```
<field
  id="vhCost"
  name="Cost Range"
  type="static"
  maxlength="20">
  </field>
```

Once we've got the new field in place, we need to integrate it into the standard display handling of the cost for gear. This entails using it appropriately within the Finalize script for the "grCost" field. Within this script, if the gear is a vehicle, we need to retrieve the cost for display from our special "vhCost" field instead of the normal "grCost" field. However, we only do this for non-military vehicles. The solution is to add a few lines of code to the script that handle this appropriately, which results in the insertion of the following code immediately after the we check for a military cost.

```
~if this is a vehicle, pull the cost from the special vehicle field
 if (tagis[component.Vehicle] <> 0) then
   @text = field[vhCost].text
   done
   endif
```

There is one remaining detail we need to address. The "vhCost" field can be up to 20 characters long, while the maximum length of the finalized value for the "grCost" field is limited to 10 characters. Consequently, we need to increase the maximum size of the finalized text to accommodate the larger size of the "vhCost" field.

Description Output

Since vehicles have their own custom fields, we need to handle them specially when synthesizing detailed description text for display. So open up the file "procedures.dat" and locate the Descript procedure. The first thing we need to do is specify that the vehicle details are handled in a separate procedure. In the code where each component type is handled separately, add the lines shown below to invoke the proper procedure for vehicles.

```
elseif (tagis[component.Vehicle] <> 0) then
  call InfoVeh
```

Once this is done, we now need to define the appropriate procedure. Scroll down a bit further in the file and add a "InfoVeh" procedure that properly synthesizes the output for the new fields we added for vehicles. We want to keep the format looking similar to the one used in the book, so we'll combine the acceleration and top speed on one line, as well as combining the toughness and armor. We also need to include climb only for aircraft. The new procedure should look similar to the one shown below.

```
<procedure id="InfoVeh" context="info"><![CDATA[
  ~declare variables that are used to communicate with our
caller and for temporary use
  var iteminfo as string

  ~report the acceleration and speed
  iteminfo = "Acc/Top Speed: " & field[vhAccel].text & "/" &
field[vhTopSpeed].text & "{br}"

  ~report the climb for aircraft
  if (tagis[VehType.Aircraft] <> 0) then
    iteminfo = "Climb: " & field[vhClimb].text & "{br}"
    endif

  ~report the toughness and armor
  iteminfo &= "Toughness (Armor): " & field[vhTough].text & " ("
& field[vhArmor].text & "){br}"

  ~report the crew size
  iteminfo &= "Crew: " & field[vhCrew].text & "{br}"
  ]]></procedure>
```

Selecting Vehicles Via Gear Tab

The final step we need to deal with is making vehicles accessible to the user. We'll add vehicles to the "Gear" tab, since vehicles are in many ways more like standard gear in behavior and the "Armory" tab is already pretty packed with material. Our basic plan will be to clone the existing portal and templates for handling gear, then adapt them for use with vehicles.

The first thing we need to do is create the new portal for showing the vehicles in a table. We can copy the existing "grGear" portal and then modify it to suit our purposes. In addition to changing the portal id, we need to change the template ids, the tag expression governing the contents managed, and the various scripts that must specifically reference "vehicles" now. The result should be something similar to the portal below.

```
<portal
  id="grVehicle"
  style="tblNormal">
  <table_dynamic
    component="Gear"
    showtemplate="grVehPick"
    choosetemplate="grVehThing"
    choosesortset="Vehicle"
    buytemplate="BuyCash"
    selltemplate="SellCash"
    descwidth="300">
    <list>component.Vehicle</list>
    <titlebar><![CDATA[
      @text = "Select Vehicles to Purchase from the List Below"
      ]]></titlebar>
    <headertitle><![CDATA[
      @text = "Miscellaneous Vehicles"
      ]]></headertitle>
    <additem><![CDATA[
      @text = "Add New Vehicle"
      ]]></additem>
  </table_dynamic>
</portal>
```

The next step is to define the template used for showing vehicles as things. This template is virtually identical to the one for standard gear. So we can clone the "grGrThing" template and adapt for our purposes very easily, resulting in the template shown below.

```
<template
  id="grVehThing"
  name="Vehicle Thing"
  compset="Vehicle"
  marginhorz="3"
  marginvert="2">

  <portal
    id="name"
    style="lblNormal">
    <label field="name">
      </label>
    </portal>

  <portal
    id="cost"
    style="lblNormal">
    <label>
      <labeltext><![CDATA[ @text = field[grCost].text
]]></labeltext>
      </label>
    </portal>

  <position><![CDATA[
    ~set up our dimensions, with a width that we dictate
    height = portal[name].height
    width = 250

    ~if this is a "sizing" calculation, we're done
    if (issizing <> 0) then
      done
      endif

    ~position the cost portal on the far right
    perform portal[cost].alignedge[right,0]

    ~position the name on the left and let it use all available
space
    portal[name].left = 0
    portal[name].width =
minimum(portal[name].width,portal[cost].left - 10)
    ]]></position>
```

```
</template>
```

We still need to setup a template for showing vehicles as a pick. Vehicles are a special type of gear that lack a variety of standard gear behaviors. As such, we can copy the "grGrPick" template and strip it down to get what we want. For example, vehicles are namable by the user and are always top-level containers. After we strip out the pieces we don't need, we end up with a template that looks like the one below.

```
<template
  id="grVehPick"
  name="Vehicle Pick"
  compset="Vehicle"
  marginhorz="3"
  marginvert="3">

 <portal
   id="name"
   style="lblNormal"
   showinvalid="yes">
   <label
     field="name">
     </label>
   </portal>

 <portal
   id="container"
   style="imgNormal">
  <image_literal
    image="container.bmp"
    istransparent="yes">
    </image_literal>
  <mouseinfo mousepos="middle+above"><![CDATA[
    call InfoHolder
   ]]></mouseinfo>
  </portal>

<portal
  id="info"
  style="actInfo">
  <action
    action="info">
    </action>
  <mouseinfo mousepos="middle+above"/>
  </portal>

<portal
  id="delete"
  style="actDelete"
  tiptext="Click to delete this vehicle">
  <action
    action="delete">
    </action>
  </portal>

<position><![CDATA[
  ~set up our height based on our tallest portal
  height = portal[info].height

 ~if this is a "sizing" calculation, we're done
  if (issizing <> 0) then
    done
    endif

 ~center the portals vertically
  perform portal[info].centervert
  perform portal[name].centervert
  perform portal[delete].centervert
  perform portal[container].centervert

  ~position the delete portal on the far right
  perform portal[delete].alignedge[right,0]
  ~position the info portal to the left of the delete button
  perform portal[info].alignrel[rtol,delete,-8]

 ~position the name on the left and let it use all available
space
  var limit as number
  limit = portal[info].left - portal[container].width – 5
  portal[name].left = 0
  portal[name].width = minimum(portal[name].width,limit)

 ~show the 'container' icon to the right of the name
  portal[container].left = portal[name].right + 5

 ~if the gear can't be deleted (i.e. it's been auto-added instead
of user-added,
  ~set the style to indicate that behavior to the user
  if (candelete = 0) then
    perform portal[name].setstyle[lblAuto]
   endif
  ]]></position>

 </template>
```

The final step is to add the table portal to the layout. The current layout contains a single portal, so the logic is trivial. Now that we're adding a new portal, we need to introduce appropriate logic to handle things cleanly. We'll position vehicles beneath standard gear, and we'll always make sure that we leave room to show at least two vehicles. Then we allow the standard gear to use as much of the remaining space is available. If the standard gear doesn't use all the space, we wrap up by having the vehicles use the remaining space. The net result is a layout that looks like the one below.

```
 <layout
   id="gear">
   <portalref portal="grGear" taborder="10"/>
   <portalref portal="grVehicle" taborder="20"/>

  <!-- This script sizes and positions the layout and its child
visual elements. -->
  <position><![CDATA[
    ~set all tables to span the full width of the layout
    portal[grGear].width = width
    portal[grVehicle].width = width

   ~position the vehicles table at the bottom with a minimum
height of 2 items
    portal[grVehicle].maxrows = 2
    portal[grVehicle].top = height - portal[grVehicle].height

   ~position and size the gear table to fill all remaining space
    portal[grGear].top = 0
    portal[grGear].height = portal[grVehicle].top - 10

   ~position and size the vehicle table to use the remaining
space at the bottom
    portal[grVehicle].top = portal[grGear].bottom + 10
    portal[grVehicle].height = height - portal[grVehicle].top
    ]]></position>

  </layout>
```

Adding Basic Vehicles

Now that we've got all of the mechanics in place for basic vehicles, we can define them. Since there can be lots of vehicles, we'll put them in their own data file, which we'll call "thing_vehicles.dat". Either copy an existing data file and gut it of all things or create the file from scratch and setup the appropriate XML basics.

Once the basic file framework is in place, we can begin defining the various vehicles. A few samples are shown below that can be used as a reference. Note that each has the minimum cost specified as the "grCost" field value. Appropriate tags are assigned to reflect the proper behavior and characteristics of each vehicle.

```
<thing
  id="vhCarriage"
  name="Horse and Carriage"
  compset="Vehicle"
  description="Description goes here">
  <fieldval field="grCost" value="1000"/>
  <fieldval field="vhAccel" value="Half-Pace"/>
  <fieldval field="vhTopSpeed" value="Pace+Running"/>
  <fieldval field="vhTough" value="10"/>
  <fieldval field="vhArmor" value="2"/>
  <fieldval field="vhCrew" value="1+3"/>
  <fieldval field="vhCost" value="$1-3,000"/>
  <usesource source="TimeMedi"/>
  <usesource source="TimePowder"/>
  <tag group="VehType" tag="Ground"/>
  <tag group="VehEra" tag="Civilian"/>
  <tag group="thing" tag="holder_top"/>
  </thing>

<thing
  id="vhSUV"
  name="Sport Utility Vehicle"
  compset="Vehicle"
  description="Description goes here">
  <fieldval field="grCost" value="20000"/>
  <fieldval field="vhAccel" value="20"/>
  <fieldval field="vhTopSpeed" value="40"/>
  <fieldval field="vhTough" value="14"/>
  <fieldval field="vhArmor" value="3"/>
  <fieldval field="vhCrew" value="1+7"/>
  <fieldval field="vhCost" value="$20-60,000"/>
  <fieldval field="vhSpecial" value="Luxury Features"/>
  <usesource source="TimeModern"/>
  <tag group="VehType" tag="Ground"/>
  <tag group="VehEra" tag="Civilian"/>
  <tag group="Vehicle" tag="AirBags"/>
  <tag group="Vehicle" tag="4WD"/>
  <tag group="thing" tag="holder_top"/>
  </thing>

<thing
  id="vhBellJet"
  name="Bell Jet Ranger"
  compset="Vehicle"
  description="Description goes here">
  <fieldval field="grCost" value="830000"/>
  <fieldval field="vhAccel" value="20"/>
  <fieldval field="vhTopSpeed" value="50"/>
  <fieldval field="vhClimb" value="20"/>
  <fieldval field="vhTough" value="11"/>
  <fieldval field="vhArmor" value="2"/>
  <fieldval field="vhCrew" value="2"/>
  <fieldval field="vhCost" value="$830,000"/>
  <usesource source="TimeModern"/>
  <tag group="VehType" tag="Aircraft"/>
  <tag group="VehEra" tag="Civilian"/>
  <tag group="thing" tag="holder_top"/>
  </thing>
```

Complex Vehicles (Savage)

Now that all the basics of vehicles are in place, we can look at how to support more complex vehicles, such as military vehicles with weapons. Each such vehicle needs to be managed as a user-selectable pick, plus the weapons need to be associated directly with each vehicle. We also need to handle the appropriate display of the weapons for each vehicle, which can be handled a few different ways.

Assigning Equipment to Vehicles

The biggest wrinkle posed by complex vehicles is how to define the various equipment separately (e.g. weapons) and then associate that equipment with the vehicle. Your first thought might be to use a bootstrap to associate the equipment, but that won't work. The vehicle is assigned to the character, so any things bootstrapped by the vehicle will also be assigned to the character. What we need is to treat each vehicle as its own container, into which the various equipment picks can be added.

In order to accomplish this, we must use a entity. When the entity is added to the character, it becomes a gizmo, which is a separate container. We can then add the various equipment picks into the gizmo. Entities are defined separately and then added via a thing. When the thing is added to the character as a pick, the associated entity is automatically added as a gizmo.

So our first task is to define an entity that we can use with vehicles. When we define the entity, we can automatically assign things into the entity, which will be added to every gizmo that derives from the entity. However, every vehicle is unique and there is nothing that must exist for every vehicle. We can also assign tags to every entity that will always be assigned to every derived gizmo, but we don't need any of those either.

This leaves us with an incredibly simple entity. The entity is merely a shell that will be separately customized for every vehicle. Since the entity is used to contain the various equipment possessed by each vehicle, we'll refer to it as the "load-out" for a vehicle and name it accordingly. This results in an entity definition that looks like the one below, which we can define at the bottom of the file "equipment.str".

```
<entity    id="LoadOut">
  </entity>
```

Putting the Entity to Use Now that we've got an entity, we need to put it to use. To demonstrate how this works, we'll define a the "A6M Zero" aircraft that is presented in the core rulebook. We start with the basic details for the vehicle, which should look like the following.

```
<thing
  id="vhA6MZero"
  name="A6M Zero"
  compset="Vehicle"
  description="Description goes here">
  <fieldval field="grCost" value="0"/>
  <fieldval field="vhAccel" value="20"/>
  <fieldval field="vhTopSpeed" value="140"/>
  <fieldval field="vhTough" value="12"/>
  <fieldval field="vhArmor" value="2"/>
  <fieldval field="vhCrew" value="1"/>
  <fieldval field="vhCost" value="0"/>
  <usesource source="TimeModern"/>
  <tag group="VehType" tag="Aircraft"/>
  <tag group="VehEra" tag="WWII"/>
  <tag group="thing" tag="holder_top"/>
  <tag group="User" tag="Military"/>
  </thing>
```

The Zero has two pairs of weapons in its armament. It has two 7.7mm machine guns and two 20mm cannons. So we next need to

define the two weapons appropriately. These should look like is shown below.

```
<thing
  id="vw77MG"
  name="7.7mm MG"
  compset="Ranged"
  description="Description goes here">
  <fieldval field="wpDamage" value="2d8+1"/>
  <fieldval field="wpShort" value="24"/>
  <fieldval field="wpMedium" value="48"/>
  <fieldval field="wpLong" value="96"/>
  <fieldval field="wpPiercing" value="2"/>
  <fieldval field="wpFireRate" value="3"/>
  <fieldval field="wpShots" value="500"/>
  <fieldval field="wpAmmo" value="7.7mm"/>
  <usesource source="TimeModern"/>
  <tag group="Equipment" tag="Natural"/>
  </thing>

<thing
  id="vw20mm"
  name="20mm Cannon"
  compset="Ranged"
  description="Description goes here">
  <fieldval field="wpDamage" value="3d8"/>
  <fieldval field="wpShort" value="50"/>
  <fieldval field="wpMedium" value="100"/>
  <fieldval field="wpLong" value="200"/>
  <fieldval field="wpPiercing" value="4"/>
  <fieldval field="wpFireRate" value="3"/>
  <fieldval field="wpShots" value="60"/>
  <fieldval field="wpAmmo" value="20mm"/>
  <usesource source="TimeModern"/>
  <tag group="Equipment" tag="Natural"/>
  <tag group="Weapon" tag="HvyWeapon"/>
  </thing>
```

We can now assign the load-out entity to the vehicle. Once we have the entity to contain the weapons, we can then assign the weapons into it. This is accomplished by bootstrapping the weapons into the entity , as opposed to bootstrapping them onto the vehicle. So we add the following material to our Zero, which adds the entity and then puts the two weapons into it.

```
<child entity="LoadOut">
  <bootstrap thing="vw77MG">
   </bootstrap>
  <bootstrap thing="vw20mm">
   </bootstrap>
  </child>
```

That's all there is to it. The two weapons now reside within the entity and behave as children of the vehicle itself.

OOPS! WE'VE GOT A PROBLEM

Go to the "Gear" tab and add a vehicle. Our new Zero shows up nicely in the list of vehicles. However, the moment that we add the Zero to the character, HL presents us with error messages. It tells us that the "live state" of the gizmo is being tested before the live state of the parent pick is resolved. Aside from that error, everything appears to be working correctly, so we need to figure out what's going wrong.

The error message is being shown twice. Curiously, we have two weapons that we are bootstrapping into the gizmo. And the gizmo is the source of the error. It turns out that we've introduced an interesting timing issue within our data files. In order to fix this, let's look a little more closely at how things work within the HL engine.

All picks have a "live" state. If the pick is "live", then it behaves normally. If it is non-live, then it is treated as if it doesn't exist within the character. Any scripts that are defined for the pick are ignored, and any scripts that try to access to the pick (e.g. to pull a value from it) report a run-time.

The live state is resolved separately for every pick. It can also be resolved at a completely different time for each pick. Depending on how the data files are structured, one pick could have its live state resolved during the Initialize phase at priority 1000 and another pick could have its live state resolved during the Final phase at priority 6000. The key requirement is that all tests governing the live state for a given pick must be performed before any scripts (for example, eval rules or eval scripts) are invoked that manipulate that pick.

There can be multiple tests governing the live state for a particular pick. In addition, the use of Secondary and Existence tag expressions (assigned to picks added through particular tables) introduces situations where additional tests are added dynamically to picks, based on how and where they are added to the character. This can make things somewhat complex for an author to keep track of, so the HL engine does what it can to simplify the situation. This amounts to resolving the live state at the latest possible time for each pick. The live state for a pick is resolved **immediately before the very first script** (whether a component script or a thing script) is scheduled on that pick. This ensures that the window of opportunity for scheduling Secondary and Existence tag expressions on the pick is maximized without introducing timing problems.

Of course, there are exceptions to every rule. For this particular behavior, the exception is for picks that attach gizmos or minions. Such picks always have their live state resolved immediately after the last test governing their live state - i.e. **as early as possible**. However, if a pick with a gizmo or minion does not possess any tests governing its live state, then its live state is resolved as normal, just before its first script.

When gizmos and minions are employed, the gizmo/minion is only live if the pick attaching it is live. And if the gizmo/minion is non-live, then none of the picks within the gizmo/minion will be live. Consequently, the live state of a pick within a gizmo must first verify that the pick attaching the gizmo is also live. This means that the live state of the pick attaching the gizmo must be resolved before the live state of picks within the gizmo are resolved.

With all that in mind, let's look back at our current problem. The error is saying that the live state of the gizmo is being tested before the live state of the pick that attaches the gizmo has been set. The live state of the gizmo is being tested by the weapon picks within it. So we need to analyze when the live state is being verified for the weapons and make sure that the live state is resolved prior to that time for the pick that attaches the gizmo.

We'll start by determining when the live state is being verified by the weapons. The weapons are always live, so the live state of the weapon picks is being resolved at the timing of their earliest script. Go the "Develop" menu and into the "Floating Info Windows" sub-menu, then select the "Show Task List (Active Hero)" option. Enlarge the window so you can see the list of tasks that are scheduled for the character. At the very top, you'll see "Pick Condition" entries for the weapons on the Zero, immediately followed by Eval scripts on those weapons. The "Pick Condition" task is the one that resolves the final live state for a pick. Just as we expected, the live state for the weapons is being resolved

immediately before the first script on those weapons. This is occurring in the Initialize phase at priority 1000.

Now let's review the timing for the vehicle itself - the pick that attaches the gizmo. Just like with the weapons, there are no tests governing the live state for the vehicle. This means that the live state for the vehicle is being dictated by the timing of the first script on the vehicle. Scrolling down through the task list, we'll find a "Pick Condition" test for the "A6M Zero" pick at a timing of Setup/10000. Just below this task, we'll see that the first Eval script for the vehicle is scheduled at the same timing.

So now we understand why the problem is occurring. The next step is to figure out how to solve it. One solution would be to move the timing of the weapon scripts to later. Another solution would be to move the timing of the vehicle scripts to earlier. However, both of these approaches could lead to a domino effect with the timing of other scripts and behaviors, so we should only change them if we have no other choice.

Let's look back at the logic used by the engine for picks that attach gizmos and minions. For those two situations, the live state is resolved immediately after the last test governing the live state. There is currently no such test for the vehicle, because none is needed, but this is the reason the vehicle's live state is resolved so late. But if we did have a live test that occurred before Initialize/1000, then the live state would be resolved then and all of our problems would disappear.

So the solution is to add test of the live state that always succeeds for the vehicle and schedule it prior to Initialize/1000. This can be accomplished by defining a ContainerReq tag expression on the Vehicle component with an expression of "TRUE" (meaning the test will always succeed). Since we now have a test of the live state on the vehicle, the live state will be resolved immediately after this test is performed, and the timing errors will go away. You can add this to the component within the file "equipment.str", and it should look like the following.

```
<containerreq phase="Initialize"
priority="900">TRUE</containerreq>
```

Once the change is made, try reloading the data files and add the Zero to the character. No errors appear. If you look at the task list, you'll see the new "Component Condition" test scheduled at Initialize/900, followed by the "Pick Condition" test scheduled right after it. The "Pick Condition" tests for the weapons follow at Initialize/1000, but everything has been safely resolved, so no error is reported.

Modify the Description Procedure

Let's take this opportunity to revise the "InfoVeh" procedure that synthesizes the details for each vehicle. We need to add the particulars the weapons within the gizmo. So open up the file "procedures.dat" and locate the procedure.

The first thing we need to do is differentiate between a vehicle that possesses a load-out entity and one that doesn't. Basic vehicles don't need the entity, so we don't define one for them. This means that we need to avoid trying to access a non-existent entity for vehicles that lack them. Within our script, we check if an entity exists and then bail out if we don't have one. The code should look something like the snippet below.

```
  ~if there is no child entity/gizmo, then there's nothing more to
do
  if (isentity = 0) then
   done
   endif
```

The next thing we need to do is report the basic load-out for the vehicle based on the contents of the entity. We want to identify each piece of equipment within the entity, list each along with the pertinent characteristics of the equipment. This can be achieved through the use of a "foreach" statement.

At this point, we run into an important distinction. Accessing the contents of a entity associated with a thing is very different from accessing the contents of gizmo beneath a pick. This is because the entity has not been added to the character yet, and that imposes some significant limitations on what can be accessed. We must use two completely different methods for accessing the contents of entities and gizmos, although the basic logic can be similar.

The problem we face is that the "wpNotes" field for the weapons is synthesized via an Eval script. No scripts are invoked for things, so we will have to synthesize all the information manually if we want to display it for things. In addition, we're going to run into some additional limitations associated with customizing things within the entity in just a moment. Since some weapons will be re-used with slight differences (e.g. ammunition quantities), we'll want to override the values of various fields via the bootstrap, but that information isn't accessible on the thing - only once the weapon is actually added to the gizmo as a pick. So we need to re-think our approach at this point.

Handling Display of Entities

Let's reconsider exactly what information we need to display to the user during selection of vehicles. The load-outs for each vehicle in Savage Worlds is fixed, and the number of vehicles is not exhaustive. Consequently, subtle differences between the load-outs of two vehicles is not going to be a serious consideration for players when selecting vehicles. That means that we really just need to identify the weapons and equipment for each vehicle by name during selection, without displaying all of the weapon characteristics. That information will be useful during the game, but not important during selection.

Given this situation, the best solution is probably to simply add a new field to each vehicle that lists equipment comprising the load-out. This field can be displayed within the description for the vehicle during selection and ignored when we have full access to the gizmo contents. So we'll define a new "vhLoadout" field for exactly this purpose, which should look like the field specification below.

```
<field
  id="vhLoadout"
  name="Load-Out"
  type="static"
  maxlength="250">
  </field>
```

Return to the Description Procedure

Now that we have the issue of load-outs resolved during selection, we can get the description procedure properly into place. If we have a thing, we will simply output the "vhLoadout" field for the vehicle.

If we have a pick, then we'll go through the contents of the gizmo and properly synthesize all of the appropriate details.

The net result is code that looks like below. This new logic can be added at the end of the "InfoVeh" procedure, after we've verified that we indeed have an entity for the vehicle.

```
 ~if this is a thing, report the basic load-out for the vehicle based
on the field;
 ~otherwise, synthesize any load-out for the vehicle as a pick
appropriately
 ~Note: We must differentiate between a thing and a pick when
accessing the entity,
 ~       as we want summary details for a thing and complete
details for a pick
 var loadout as string
 if (ispick = 0) then
  loadout = field[vhLoadout].text & "{br}"
 else
  loadout = ""    foreach pick in gizmo
   loadout &= "{horz 10}" & chr(149) & " " &
eachpick.field[name].text & ": "
   loadout &= eachpick.field[wpDamage].text & " - " &
eachpick.field[wpRange].text
   if (eachpick.field[wpNotes].isempty = 0) then
    loadout &= " - " & eachpick.field[wpNotes].text
    endif
   loadout &= "{br}"
   nexteach
  endif

 ~if there is any load-out for the vehicle, output it
 if (empty(loadout) = 0) then
  iteminfo &= "Weapons/Equipment:{br}" & loadout
  endif
```

Refining the Entity

Let's return at our Zero now. We need to specify suitable text for the new "vhLoadout" field. For clarity, we'll put each weapon on a new line and indent it. This yields the following contents for the field.

```
 <fieldval field="vhLoadout" value="   2x 7.7mm MGs{br}   2x
20mm Cannons"/>
```

Now we'll take a look at our Zero within HL. Go to the "Gear" tab and add a vehicle. Our Zero should be listed, and it should show the load-out equipment properly. Add the Zero to the character, then check the mouse-info text for the vehicle. The display is different, since the Zero actually has **two** of each weapon that is listed. We need to fix this.

Before we continue, let's take a look at various other vehicles and their load-outs. Some weapons are re-used for different vehicles, which is great, but different ammunition quantities are specified. So we have to handle that. In addition, some vehicles simply re-use weapons and tweak them slightly. We can either implement lots of different weapons that are nearly identical or customize each weapon when it is added to the vehicle.

All of the adjustments can be most easily handled by customizing each weapon via the "bootstrap" element that adds it to a vehicle. Each bootstrap element can possess optional "assignval" child elements, which allow you to override the value of individual fields on the child pick. We can use "assignval" to modify the name, ammunition quantity, or any other facet of individual weapons.

In order to override the values of fields, those fields must be able to be overridden. This requires that a field be designated as the "derived" type. However, the majority of the fields used on weapons are "static". So we need to identify the various fields that we'll want to override and change them from "static" to "derived". After looking through the various vehicles in the core rulebook, the list of fields we need to change include: "wpPiercing", "wpFireRate", "wpShots", and "wpAmmo". Open the file "equipment.str" and modify each of these fields to the new type.

Once this is done, we can customize the weapons that are bootstrapped into our Zero appropriately. For each weapon, we can tailor the name via the "livename" field and we can tailor the ammunition quantity via the "wpShots" field. This results in a revised "child" element for the Zero that looks like below.

```
<child entity="LoadOut">
  <bootstrap thing="vw77MG">
   <assignval field="livename" value="2x 7.7mm MGs"/>
   <assignval field="wpShots" value="500"/>
   </bootstrap>
  <bootstrap thing="vw20mm">
   <assignval field="livename" value="2x 20mm Cannons"/>
   <assignval field="wpShots" value="60"/>
   </bootstrap>
  </child>
```

Reload the revised data files and add a Zero to the character again. It will now display its load-out details with appropriate names and accurate ammunition quantities.

Refinement of Behaviors

Using Bitmaps for Die Types (Savage)

The contents of the attribute and skill incrementers currently show the die-type and any adjustment using the format "d6+2". It would be ideal if we actually showed a bitmap for each die-type, since it would tailor things much more closely to how Savage Worlds works. Depending on what we ultimately want to achieve, doing this is either easy or a small amount of work.

The Easy Solution

The Skeleton data files include an assortment of bitmaps for the various die-types. There are bitmaps for use on the screen and within character sheet output. All of these bitmaps are automatically copied into place for your use when you create a new game system. All you need to do is reference them.

We've already centralized all of the handling for what gets displayed into a single place. The field "trtDisplay" contains the text to be output, and the procedure "FinalRoll" handles synthesizing this text. So all we need to do is modify the "FinalRoll" procedure to incorporate the bitmaps instead of using simple text.

Open the file "procedures.dat" and locate the "FinalRoll" procedure. Within this procedure, there is a line that generates the die-type for display by combining the letter "d" with the numeric value. We can easily change this to use the appropriate bitmap instead.

You can insert the bitmap into the text via the use of encoded text. The syntax for inserting a bitmap via encoded text is "{bmp filename}", where filename is the base name of the file to be inserted. Since the die-type bitmaps intended for use on screen are named in the form "d6_screen.bmp", the corresponding encoded text for a d6 should be "{bmp d6_screen}". This means that we can

change one line of script code and get the die-type appearing as a bitmap. The new line of code should be the following.

```
finaltext = "{bmp d" & dietype & "_screen}"
```

Reload the data files and see what you think. The die-type uses the proper bitmap and any adjustment is appended as a text value. Not bad for a single line of code being changed, but we can probably do a bit better.

Refining the Behavior

As expected, after a little bit of testing, the results aren't quite optimal. So let's try refining the handling a bit. For example, if there is an adjustment made to the final roll (e.g. "d6+2"), the "+2" is a bit too small now. We'll start by increasing the font size.

The incrementers for showing die-types all use the "incrDie" style, so we'll need to modify it. Open up the file "styles_ui.aug" and locate the style. It currently uses the "fntincrsim" font resource, which is also used by another incrementer, so we can't simply change it. Instead, we need to define a new font resource, which we'll call "fntincrdie". We can define the new resource as part of the style, and we'll pick a size like 54 as a good balance that hopefully won't overshadow the die-type bitmap. This yields a revised style definition like the one below.

```
<style
  id="incrDie">
  <style_incrementer
    textcolor="f0f0f0"
    font="fntincrdie"
    editable="no"
    textleft="13" texttop="0" textwidth="44" textheight="20"
    fullwidth="70" fullheight="20"
    plusup="incplusup" plusdown="incplusdn" plusoff="incplusof"
    plusx="59" plusy="0"
    minusup="incminusup" minusdown="incminusdn"
minusoff="incminusof"
    minusx="0" minusy="0">
    </style_incrementer>
  <resource
    id="fntincrdie">
    <font
      face="Arial"
      size="54">
      </font>
    </resource>
  </style>
```

If we add the Ace edge and the Boating skill, we can see what this looks like. Unfortunately, this new size is a little too prominent compared to the die-type bitmap, so we need to make it a little less white. This is achieved by modifying the "textcolor" attribute within the style and dialing down the brightness a little bit. We'll pick a value of "c0c0c0" as a good compromise.

This still doesn't look quite right. The "+" looks good, but the numeric adjustment looks a little too frail. We can change the font resource to be bold, but that results in the "+" looking blocky and ugly. What we need is for the "+" to be non-bold and the numeric value to bold. We can accomplish this by revising the procedure a little bit to carve up the bonus into two pieces. The net result is a revised procedure script that looks like below.

```
~declare variables that are used to communicate with our caller
 var finaldie as number
 var finaltext as string

~bound our final die type appropriately
 var final as number
 final = finaldie  if (final < 2) then
   final = 2  elseif (final > 6) then
   final = 6
   endif

~convert the final value for the trait to the proper die type for
display
 var dietype as number
 dietype = final * 2  finaltext = "{bmp d" & dietype & "_screen}"

~if there are any bonuses or penalties on the roll, append those
the final result
 if (finalbonus <> 0) then
   var bonus as string
   bonus = signed(finalbonus)
  finaltext &= left(bonus,1) & "{b}" & right(bonus,1)
   endif
```

We now have something that works a bit better, but it still doesn't look great. When some traits have adjustments and others don't the die-type bitmaps seem to bounce around due to the centering logic we're using. In addition, having the adjustment value aligned at the baseline of the bitmap isn't very visually appealing. More importantly, the changes we've introduced have caused a number of cascading issues elsewhere, such as the damage and description text for weapons, which are now mucked up.

A Different Approach

Sometimes it's better to step back and re-consider the approach you're taking. In this case, we can continue trying to refine the behavior to get something we like, but it's doubtful we'll end up with something truly good. So we're going to abandon our current approach and put everything back the way it was when we started. This includes both the "incrDie" style and the "FinalRoll" procedure.

Instead of trying to squeeze both the die-type bitmap and the adjustment into the incrementer, let's re-think what we need. The incrementer controls the die-type and nothing more. Any adjustment is completely distinct from the die-type and does not need to be incorporated into the incrementer contents. In fact, it would be perfectly reasonable to move the adjustment out of the incrementer and have it appear next to the incrementer. So that's the approach we'll use this time.

In order to use this approach, we'll need to synthesize two different pieces of text for each trait. We'll need the current "trtDisplay" field that contains a text version we can use in places where we don't want to include the die-type bitmaps. For example, the description text looks better without the die-type bitmap. We'll also need a separate field with the die-type bitmap that we can show within the incrementers.

This means we need to define a new field for the die-type bitmap and integrate it appropriately. First, we define the field, which we'll call "trtIncr" due to its intended use with the incrementers. The field should look like the one defined below.

```
<field
  id="trtIncr"
  name="Die-Type for Incrementers"
  type="derived"
  maxlength="50">
```

```
</field>
```

The next step is to properly synthesize the field contents. We can augment both the "FinalRoll" procedure and the Eval script that calls the procedure to achieve this. A new parameter can be passed into the procedure that is returned with the appropriate bitmap information, and that parameter can then be assigned to the new field. The revised procedure and Eval script should look similar to the ones shown below.

```
 <procedure id="FinalRoll" scripttype="none"><![CDATA[
   ~declare variables that are used to communicate with our
caller
   var finaldie as number
   var finalbonus as number
   var finaltext as string
   var dietext as string

  ~bound our final die type appropriately
   var final as number
   final = finaldie
   if (final < 2) then
     final = 2
   elseif (final > 6) then
     final = 6
     endif

  ~convert the final value for the trait to the proper die type for
display
   var dietype as number
  dietype = final * 2
   finaltext = "d" & dietype

  ~if there are any bonuses or penalties on the roll, append
those the final result
   if (finalbonus <> 0) then
     finaltext &= signed(finalbonus)
     endif

  ~convert the final value for the trait to the proper die type
bitmap
   dietext = "{bmp d" & dietype & "_screen}"
   ]]></procedure>

 <eval index="4" phase="Render" priority="5000" name="Calc
trtDisplay">
   <after name="Calc trtNetRoll"/>
   <after name="Calc trtFinal"/><![CDATA[
   ~if this is a derived trait, our display text is the final value
   if (tagis[component.Derived] <> 0) then
     field[trtDisplay].text = field[trtFinal].value
     done
     endif

  ~calculate the net bonuses and penalties for the roll
   var finalbonus as number
   finalbonus = field[trtNetRoll].value

  ~generate the appropriate results for display
   var finaldie as number
   var finaltext as string
   var dietext as string
   finaldie = field[trtFinal].value
   call FinalRoll

  ~put the final results into the proper fields
   field[trtDisplay].text = finaltext
   field[trtInfo].text = dietext
   ]]></eval>
```

At this point, our new field is ready to go and we simply need to put it to use within the incrementers. That's a simple step. All you need to do is locate the "trtUser" field that drives the values shown within the incrementer and revise the "Finalize" script to pull the text from the "trtIncr" field instead of "trtDisplay". Once that's done, we can see how things look. This new approach looks like it could be a good bit better, so we'll continue refining it.

Refining the New Approach

The incrementer was sized to accommodate the adjustment, which is no longer being included. So we need to revise the incrementer style we use for showing the traits. Open the file "styles_ui.aug" and locate the "incrDie" style that we worked with previously.

This time, we need to adjust the overall width of the incrementer. That requires we also adjust the width of the text region in the center, as well as the offset of the arrow for increasing the value. If we drop the overall width to about 58 pixels, everything should look pretty good. This yields a new style definition like the one below.

```
 <style
   id="incrDie">
   <style_incrementer
    textcolor="f0f0f0"
    font="fntincrsim"
    editable="no"
    textleft="13" texttop="0" textwidth="32" textheight="20"
    fullwidth="58" fullheight="20"
    plusup="incplusup" plusdown="incplusdn" plusoff="incplusof"
    plusx="47" plusy="0"
    minusup="incminusup" minusdown="incminusdn"
minusoff="incminusof"
    minusx="0" minusy="0">
    </style_incrementer>
   </style>
```

Where's the Adjustment?

There's one critical piece that we still haven't dealt with. The new approach omits the adjustment from the incrementer, so we need to go back and add it to the interface somewhere. The two places where we will be showing the adjustment separately are on the Basics tab for attributes and on the Skills tab for skills. We'll need to address each of these locations, but the general approach will be the same.

We'll start with the Basics tab and the attributes, which are controlled by the "baAttrPick" template. Open the file "tab_basics.dat" and locate the template. We're going to need more horizontal space to fit the adjustment value, so the first thing to do is decrease the horizontal margin to something like "10" pixels.

The next step is to add a new portal for displaying the adjustment value. The contents of this new portal will be driven by a script, and we should either show the adjustment value or an indicator that there is no adjustment for this trait. If there is no adjustment, the indicator should be faint so that it doesn't draw attention away from the important information being shown. The adjustment value can be pulled from the "trtNetRoll" field, so this should yield a new portal similar to the one below.

```
 <portal
   id="adjust"
   style="lblLarge">
   <label>
    <labeltext><![CDATA[
```

```
      if (field[trtNetRoll].value = 0) then
        @text = "{text 808080}" & chr(150)
      else
        @text = signed(field[trtNetRoll].value)
        endif
      ]]></labeltext>
    </label>
  </portal>
```

With the portal defined, we now need to integrate it into the Position script for the template. Since the adjustment can vary in contents, it can also vary in width. Consequently, we need to allocate a fixed width to the portal so that everything behaves consistently. The revised Position script should look similar to the one below.

```
 ~set up our height based on our tallest portal
 height = portal[info].height

 ~if this is a "sizing" calculation, we're done
 if (issizing <> 0) then
   done
   endif

 ~freeze our value in advancement mode or if an advancement
has modified us
  ~Note: All freezing must be done *before* any positioning is
performed.
 if (state.iscreate = 0) then
   portal[value].freeze = 1
 elseif (autonomous = 0) then
   portal[value].freeze = 1
   endif

 ~position our tallest portal at the top
 portal[info].top = 0

 ~center the other portals vertically
 perform portal[name].centervert
 perform portal[value].centervert
 perform portal[adjust].centervert

 ~position the info portal on the far right
 perform portal[info].alignedge[right,0]

 ~assume a standard width for the adjustment, since it can vary,
then position

 ~the adjustment to the left of the info portal (plus a gap)
 portal[adjust].width = 20  perform
portal[adjust].alignrel[rtol,info,-15]

 ~position the incrementer to the left of the adjustment portal
(plus a gap)
 perform portal[value].alignrel[rtol,adjust,-10]

 ~position the name on the left and make sure its width does not
exceed the available space
 portal[name].left = 0
 portal[name].width =
minimum(portal[name].width,portal[value].left - portal[name].left -
10)
```

The adjustment is too small, so try switching to the "lblXLarge" style for the portal. That's way too big. We need something inbetween, but there is no label style with a font in the size we need. We'll have to define a new style to give us what we want. Open the file "styles_ui.aug" and locate the existing "lblLarge" style. Clone it and we can adapt it to our purposes. Assign it a new id of "lblAdjust" and change the font to "fntAdjust". Then we need to define the "fntAdjust" resource as part of the style, using a font size of about 50. The net result is a style definition like the one below.

```
<style
  id="lblAdjust">
  <style_label
   textcolor="f0f0f0"
   font="fntadjust"
   alignment="center">
   </style_label>
  <resource
   id="fntadjust">
   <font
    face="Arial"
    size="50"
    style="bold">
    </font>
   </resource>
  </style>
```

Once the style is defined, change the "adjust" portal over to use the new style. Then reload and see how things look. Not bad at all.

Modify The Skills Tab

We now need to do something comparable on the Skills tab to show any adjustments. Fortunately, that's an easy task. We can start by copying over the portal from the Basics tab, after which we'll need to change the style to "lblLarge" for a more suitable font size. The portal should look like the following.

```
<portal
  id="adjust"
  style="lblLarge">
  <label>
   <labeltext><![CDATA[
    if (field[trtNetRoll].value = 0) then
      @text = "{text 808080}" & chr(150)
    else
      @text = signed(field[trtNetRoll].value)
      endif
    ]]></labeltext>
   </label>
  </portal>
```

Once the portal is in place, we need to integrate the new portal into the template via the Position script. The revised Position script should end up looking similar to the following.

```
 ~set up our height based on our tallest portal
 height = portal[info].height

 ~if this is a "sizing" calculation, we're done
 if (issizing <> 0) then    done    endif

 ~freeze our value in advancement mode or if an advancement
has modified us
  ~Note: All freezing must be done *before* any positioning is
performed.
 if (state.iscreate = 0) then
   portal[value].freeze = 1
 elseif (autonomous = 0) then
   portal[value].freeze = 1
   endif

 ~position our tallest portal at the top
 portal[info].top = 0

 ~position the other portals vertically
 perform portal[name].centervert
 perform portal[domain].centervert
```

```
  perform portal[lbldomain].alignrel[btob,domain,0]
  perform portal[value].centervert  perform
portal[delete].centervert
  perform portal[adjust].centervert

 ~position the delete portal on the far right
  perform portal[delete].alignedge[right,0]

 ~position the info portal to the left of the delete button
  perform portal[info].alignrel[rtol,delete,-8]

 ~position the incrementer on the left
  portal[value].left = 0

 ~assume a standard width for the adjustment, since it can vary,
then position
  ~the adjustment to the right of the incrementer
  portal[adjust].width = 20
  perform portal[adjust].alignrel[ltor,value,10]

 ~position the name next to the adjustment
  perform portal[name].alignrel[ltor,adjust,10]

 ~if we don't need a domain, hide it and let the name use all
available space
  if (tagis[User.NeedDomain] = 0) then
    portal[lbldomain].visible = 0
    portal[domain].visible = 0
    portal[name].width =
minimum(portal[name].width,portal[info].left - portal[name].left -
10)

 ~otherwise, position the domain portals next to the name
  else
    perform portal[lbldomain].alignrel[ltor,name,20]
    perform portal[domain].alignrel[ltor,lbldomain,5]
    portal[domain].width = 150
    endif

 ~if the ability is auto-added, change its font to indicate that fact
  if (candelete = 0) then
    perform portal[name].setstyle[lblAuto]
    endif
```

At this point, we now have the bitmaps tied in properly for use within incrementers, and the adjustments are cleanly separated and handled in a very smooth fashion.

SHOW THE DERIVATION OF VALUES (SAVAGE)

Within Savage Worlds, many traits are influenced by a variety of factors. When the user sees one of these traits on the screen, there will be times when the number just doesn't look right. Usually, this will be due to some effect being included that the user has overlooked but the data files are properly tracking. At times like this, it would be incredibly helpful to inform the user how the final trait value was actually calculated for the character, presenting each of the contributing influences so the user can see what he's overlooked. We'll take the time to add this information in a few key spots.

BUILT-IN HISTORY TRACKING

The first thing we need to do is provide a place to accrue the derivation details for each trait. Your first thought is probably to define a new text-based field where this information can be tracked, and that approach would definitely work. However, the Kit provides built-in handling for just this type of situation.

Fields that are value-based and derived can leverage history tracking that is wholly managed by HL. The tracking can take three different forms. The first is "best" tracking, which only tracks the largest adjustment applied to a given field and is ideally suited for situations where adjustments do not stack. The second is "stack" tracking, where each individual adjustment is tracked. The third is "changes" and is identical to "stack", except that it automatically ignores any adjustments that have no net effect (e.g. adding zero). We'll want to use the second form and present a report to the user that shows all of the adjustments that were applied.

We can enable history tracking very easily by adding a new attribute to the field definition. Since all of the adjustments we want to track are applied to the "trtBonus" field of Traits, we'll add history tracking to that field. The modified field definition should look similar to the one shown below.

```
<field
  id="trtBonus"
  name="Bonus Value"
  type="derived"
  history="stack">
  </field>
```

TRACKING OF ADJUSTMENTS

At this point, we could reload the data files and everything would continue to work as before. Although history tracking has been enabled for the "trtBonus" field, we can choose when to track adjustments and when not to do so. The tracking mechanism is optional, allowing us to apply adjustments that aren't tracked. This is important, because the same field may need to utilize history tracking for some things and not for others. This also makes it easy for us to integrate history tracking in a step-wise process instead of requiring us to do it all at once.

In order to track an adjustment within the history for a field, we need to utilize the "modify" target reference on the field. This target reference must be given an operator (e.g. "+" or "-"), a value, and a string that describes what the adjustment represents. Adding 2 to the "trtBonus" field for the "trCharisma" trait would look similar to the line of code below.

```
 perform hero.child[trCharisma].field[trtBonus].modify[+,2,"The
Reason"]
```

Since we're going to be using this basic statement in lots of places, we might as well make it easy on ourselves. So we'll define a new script macro for this field that works similarly to the "traitbonus" script macro that we already use whenever modifying the value of a trait. Open the file "definition.def" and locate the various script macros. Now define a new macro that takes four parameters and resolves to the "modify" usage shown above. The new macro should look like the following.

```
<scriptmacro
  name="traitadjust"
  param1="trait"
  param2="oper"
  param3="value"
  param4="text"

result="hero.child[#trait].field[trtBonus].modify[#oper,#value,#text
]"/>
```

Since a macro is simply text that gets swapped in during compilation, we could also include the "perform " statement in the macro. If we do not include it, using the macro will look like the

first example below. If we do include it, then our code will look like second example below. You are free to choose what works best for you, but we'll use the former approach as we continue this topic.

```
perform #traitadjust[trCharisma,2,"The Reason"]
```

```
#traitadjust[trCharisma,2,"The Reason"]
```

Revise Adjustments for Derived Traits

We now need to put the adjustment tracking to use within the data files. This entails identifying the places where we need to switch to the new mechanism and converting them appropriately. We'll start by focusing on the four derived traits for Savage Worlds, since these are the most heavily modified via different effects.

The first thing we need to do is setup the basic thing definitions for these traits to properly use the history tracking. Open the file "thing_traits.str" and locate the derived traits. We'll now go through them, one at a time.

The first trait is "trPace". It properly sets up the initial field value of 6, and the Eval script bounds the "trtFinal" field, so there is nothing we need to do here.

The next trait is "trParry", for which the Eval script includes both the starting value of 2 and adds half the Fighting skill. We need to change this so that we define the initial field value to be 2 via a "fieldval" element. Then we need to change the Eval script to utilize the macro to apply to adjustment for half the Fighting skill. We can put the adjustment for the Fighting skill within an "if" statement if we wish, which will only show the adjustment if there is an actual skill, but it's also valid to always include the adjustment and have it report a value of zero. These two changes are shown below.

```
<fieldval field="trtBonus" value="2"/>

if (hero.childexists[skFighting] <> 0) then
   perform field[trtBonus].modify[+,#traitfound[skFighting],"Half
Fighting"]
   endif
```

Next up is the "trCharisma" trait, which doesn't require any changes. We could be complete and specify a "fieldval" element with an initial value of 0, but that's not truly necessary.

Lastly, we have the "trTough" trait. This trait is similar to the "trParry" trait in its behavior. We need to define the initial field value of 2 via the "fieldval" element. We also need to change the Eval script to apply the proper adjustment for the Vigor attribute. These two changes are shown below.

```
<fieldval field="trtBonus" value="2"/>

perform field[trtBonus].modify[+,#trait[attrVig],"Half Vigor"]
```

We can now go through all the places in the data files where we modify the bonus for these four traits, converting each to apply the tracked adjustment. For example, within the "thing_edges.dat" file, the "Berserk" edge applies two separate adjustments to these traits. We need to change the penalty on the "trParry" trait to use the new adjustment macro and do the same for the bonus to the "trTough" trait. We can attribute both to the "Berserk" edge, yielding a new script that looks similar to the one below.

```
if (field[abilActive].value = 0) then
   perform #traitadjust[trParry,-,2,"Berserk"]
  #traitroll[skFighting] += 2
   perform #traitadjust[trTough,+,2,"Berserk"]
   endif
```

Go through the data files and locate all instances where the derived traits are being modified. Every instance should be modified to use the new "#traitadjust" macro that logs what adjustment was applied. Be sure to also track the effects of equipped weapons and shields on the Parry trait, as well as the effects of equipped armor on the Toughness trait.

Report the History for Derived Traits

The derivation history should now be getting properly synthesized for each derived trait, so all we need to do now is make that history accessible to the user. Open the file "tab_basics.dat" and locate the "baTrtPick" template that is used to display the various derived traits. Within the template, the "details" portal shows the final value for the trait, and a MouseInfo script is defined that currently just outputs "???".

We can change the script to report the adjustment history for the "trtBonus" field. This is achieved by using the "history" target reference on the field, specifying a suitable separator between each entry. We also want to be sure that the starting value is shown for each derived trait. The derivation details will now be available to the user by simply moving the mouse over the value. The revised portal should look like the one shown below.

```
<portal
  id="details"
  style="lblLarge">
  <label>
    <labeltext><![CDATA[
      @text = field[trtDisplay].text
     ]]></labeltext>
  </label>
  <mouseinfo mousepos="middle+above"><![CDATA[
    @text = field[trtBonus].history[", ",start]
   ]]></mouseinfo>
  </portal>
```

Showing Derivation for Roll Adjustments

The derivation history for roll adjustments on attributes and skills would also be extremely useful. Seeing a "+1" or "-2" next to an attribute or skill is helpful, but it's vastly more helpful to be able to quickly see what is causing that adjustment. We can utilize the same history mechanisms outlined above for this purpose as well.

The calculation of the "trtNetRoll" field is based on three separate components. One of those is the impact of wounds and/or fatigue, so history tracking is not really applicable. However, the other two facets are the non-stacking adjustments due to professions and all the other stacking adjustments that can be accrued. Each of these is perfectly suited to history tracking.

The first step is to change the two fields to utilize history tracking. The "trtRoll" field allows all the changes to be stacked, so we can institute "stack" history tracking, just like we did for the "trtBonus" field of derived traits. The "trtNoStack" field, though, does not support stacking. As such, we can utilize "best" history tracking to

automatically identify only the biggest adjustment. After putting these changes into place, we should end up with revised field definitions like below.

```
<field
  id="trtRoll"
  name="Bonus on Trait Rolls"
  type="derived"
  history="stack">
  </field>

<field
  id="trtNoStack"
  name="Professional Trait Bonus"
  type="derived"
  history="best">
  </field>
```

The next step is to setup appropriate macros to put the history mechanism to use for each field. We already have the "traitroll" and "traitprof" macros defined, so what we need to do is adapt them to use the history mechanism instead. This yields the revised macros shown below, which we'll then need to utilize differently throughout the data files.

```
<scriptmacro
  name="traitroll"
  param1="trait"
  param2="oper"
  param3="value"
  param4="text"

result="hero.childfound[#trait].field[trtRoll].modify[#oper,#value,#t
ext]"/>

<scriptmacro
  name="traitprof"
  param1="trait"
  param2="oper"
  param3="value"
  param4="text"

result="hero.childfound[#trait].field[trtNoStack].modify[#oper,#val
ue,#text]"/>
```

Before we change all the field references to use the new macros, let's first look at how everything will be shown to the user. We need to display the derivation details in two separate places - for attributes on the Basics tab and for skills on the Skills tab. These involve two separate templates, but the logic we want will be the same for both. So we'll define a single procedure that can be used from both places and setup a MouseInfo script in each place to retrieve the information.

The procedure needs to show all three facets of the derived value. So we'll setup the procedure to list each facet on a separate line, with the appropriate details for each being displayed. If there is nothing to report for a particular facet, we need to display that appropriately. Since the procedure will only be used from within a MouseInfo script, we can tailor it to directly generate the text to be displayed.

The resulting procedure is shown below. Just below that, the MouseInfo script that should be added to the "adjust" portal within both templates is also presented. The MouseInfo script calls the procedure directly, relying on the procedure to put the proper information into the "@text" special symbol that will be displayed to the user.

```
<procedure id="TrtDerive" scripttype="mouseinfo"><![CDATA[
  var info as string

  ~start with the roll adjustments that stack
  info = field[trtRoll].history[", "]
  if (empty(info) <> 0) then
    info = chr(150)
  else
    info = signed(field[trtRoll].value) & ", " & info
    endif
  @text = "Stacking Adjustments: " & info & "{br}"

  ~append the non-stacking professional adjustments
  info = field[trtNoStack].history
  if (empty(info) <> 0) then
    info = chr(150)
  else
    info &= " (" & signed(field[trtNoStack].value) & ")"
    endif
  @text &= "Professional Adjustments: " & info & "{br}"

  ~append any penalties due to wounds or fatigue
  @text &= "Wound/Fatigue Penalties: " &
signed(herofield[acNetPenal].value)
  ]]></procedure>

<mouseinfo mousepos="middle+above"><![CDATA[
  call TrtDerive
  ]]></mouseinfo>
```

We can now go through the data files and convert all uses of the "traitroll" and "traitprof" macros over to the new versions. The examples below show the properly revised Eval scripts for the "Ace" and "Strong Willed" edges.

```
perform #traitprof[skBoating,+,2,"Ace"]
perform #traitprof[skDriving,+,2,"Ace"]
perform #traitprof[skPiloting,+,2,"Ace"]

perform #traitroll[skIntimid,+,2,"Strong Willed"]
perform #traitroll[skTaunt,+,2,"Strong Willed"]
```

Don't forget that you'll also need to revise a few places where the "trtRoll" field is being used without the macro. These include the application of penalties for exceeding the load limit and the assignment of in-play adjustments. Once everything is switched over, you should be able to see all of the adjustment details by simply moving the mouse over the adjustment value within HL.

Showing Derivation for Character Creation Resources

We can use the exact same process to accrue how the user has spent the initial character creation resources and display them on the Basics tab. All character creation resources are managed via the "Resource" component, with the "resSpent" field tracking the actual resources that are consumed. So we can institute history tracking for the "resSpent" field and log the selections made by the user for subsequent reporting.

We start by modifying the field to enable history tracking, as shown below.

```
<field
  id="resSpent"
  name="Quantity Spent"
  type="derived"
  history="stack">
```

```
</field>
```

After that, we can revise the "resspent" macro to apply tracked modifications.

```
<scriptmacro
  name="resspent"
  param1="resource"
  param2="oper"
  param3="value"
  param4="text"

result="hero.child[#resource].field[resSpent].modify[#oper,#value
,#text]"/>
```

We can now go through all instances where the "resspent" macro is used and convert them appropriately. In general, we'll want to specify the name of the edge, hindrance, or whatever as the source of the adjustment. For example, arcane powers each consume one slot via an Eval script, and that script gets revised to the version shown below.

```
<eval index="2" phase="Setup" priority="5000"><![CDATA[
  perform #resspent[resPowers,+,1,field[name].text]
  ]]></eval>
```

Similarly, the accumulation of attribute points should be revised to look like the following.

```
perform #resspent[resAttrib,+,field[trtUser].value -
2,field[name].text]
```

These are all pretty simple to revise. The one that requires a little bit of thought is the handling of advances. For advances, we need to actually synthesize a truly useful name. If we simply use the type of the advance, then we'll end up with a bunch of vague references (e.g. "New Skill", "New Attribute", etc.). What we really need is both the type of advance and the details of the advance. This entails synthesizing an appropriate name, which yields a script that looks like the one below.

```
var name as string
name = field[advAction].text & ": " &
gizmo.firstchild["Advance.Gizmo"].field[name].text
perform #resspent[resAdvance,+,field[advCost].value,name]
```

There is one place where the "#resspent" macro is being used to retrieve the value and not modify it. In this instance, which deals with the load limit handling, we need to access the appropriate field value directly. The code below shows the old line and the new line that replaces it.

```
spent = #resspent[resEncumb]

spent = hero.child[resEncumb].field[resSpent].value
```

We must also identify any other places where the "resSpent" field is used directly. There happens to be one for handling the accrual of skill points, and it can be changed to use the new macro. The line of code that needs to be changes should look like below.

```
perform #resspent[resSkill,+,points,field[name].text]
```

The final step is to display the history information via a MouseInfo script. Such a script is already in place within the "details" portal in the "baResource" template on the Basics tab. So all we need to do is change the script to return the appropriate information or a suitable message if there is nothing to report. This amounts to the code below.

```
@text = field[resSpent].history[", "]
if (empty(@text) <> 0) then
  @text = "No Adjustments"
  endif
```

If we reload the data files and construct a character, we can move the mouse over the various character creation resource numbers and see a report of how those resources have been spent. However, there is a minor problem. Every attribute is logging its consumption, including attributes that are at the minimum rating of "d4" and consuming zero attribute points. Listing these attributes seems rather silly, since they don't provide any useful information for the user. Fortunately, we can easily omit them. If we go back to the "resSpent" field and modify the history tracking behavior to "changes", we'll only get adjustments that actually change something, so these adjustments of zero will be automatically thrown out.

Showing Derivation for Encumbrance and Load Limit

As a side-effect of showing the derivation of character creation resources, we've gained the ability to display the derivation of encumbrance for the character. Both the tables on the Basics tab use the same template, which means both are now hooked up to show the history within the "resSpent" field. Since the accrual of encumbrance also uses the "#resspent" macro, we had to modify it during our conversion.

The net result is that we can add some gear to a character and see the derivation without any further work. Move the mouse over the Encumbrance value shown and you should see a report of the items that contribute to the encumbrance rating for the character.

The one drawback is that we also inherit this behavior for the Load Limit value. Since there is nothing spent for this resource, moving the mouse over the value shows a message "No Adjustments". That's not ideal, but it's not horrible either.

If we really wanted to show something better for the Load Limit, we could do a lot of special handling. Alternately, we could get creative. If we setup a single adjustment using the "#resspent" macro for the resource, the text we use for the adjustment could be the text we want to show to the user via the MouseInfo script. Since we don't actually use the final value of the resource for anything, nor do we show it anywhere, spending the resource to add an entry to the history would work nicely.

If you think of something useful to report for the load limit, feel free to give this tactic a try.

Revise Summary Panels (Savage)

Everything is finally coming together nicely with our data files. At this point, it's probably a good idea to switch our focus over to the

various summary panels that are displayed.

Assess the Contents

When is comes to the summary panels, the first thing we need to do is assess what information needs to be shown within the panels. This is most easily accomplished by simply making a list of the various pieces of information that should be included. A useful approach is to look at each of the main tabs within the interface and identify items that the user will likely want to refer to while looking at other tabs. After giving it some thought, the following list probably makes the most sense.

- Attributes
- Derived Attributes
- Status Info (e.g. Encumbrance)
- Skills
- Edges
- Hindrances
- Racial Abilities
- Armor
- Hand Weapons
- Ranged Weapons
- Special Weapons

Organize the Information

The next step is to break the above list into logic groupings that will generally fit into a single summary panel. An important consideration during this step is that you need to create your data files with a small 800x600 display in mind. Even if you are using a huge display and keep the HL application at maximum size, your data files will also be used by some people on a small 800x600 display. So it's important that you keep each summary panel reasonably compact, if at all possible.

After you break things up, you can then decide on the appropriate sequence in which to show each set of information. For example, should hand weapons go above ranged weapons, or vice versa? When looking at the list of contents we identified above, there are a variety of suitable organizations, and the final determination really comes down to what you think works best. For our purposes, we'll settle on three separate summary panels that are organized as shown below.

- Basics
  - Attributes
  - Derived Attributes
  - Status Info (e.g. Encumbrance)
- Abilities
  - Edges
  - Hindrances
  - Racial Abilities
  - Skills
- Armory
  - Armor
  - Hand Weapons
  - Ranged Weapons
  - Special Weapons

The Basics Panel

We can now set about configuring the first summary panel that contains all the basics of the character. We'll use the existing "Basics" panel, which can be found in the file "summ_basics.dat". There is already a table of attributes and another table of derived traits, so we don't need to add those. However, we do need a table in which to display the encumbrance and other status information.

Adding the new table starts with cloning one of the existing two tables. Once we clone it, we can then modify it for showing status information. We can look at the "baStatus" table used on the Basics tab for guidance regarding how to setup the new table. Since the format of the output will be different from standard traits, we'll also need to create a new template, which we'll name appropriately. The new table portal should look similar to the one below.

```
<portal
  id="smStatus"
  style="tblInvisSm">
  <table_fixed
    component="Resource"
    showtemplate="smResource"
    showsortset="explicit"
    scrollable="no">
    <list>Helper.Status</list>
    </table_fixed>
  </portal>
```

With the table in place, we now need to define a suitable template. We can clone the "smTrait" template and adapt it for our needs. We need to change the component set reference and the field which shows the value. We also need to give ourselves significantly more space for the value field, which can be done within the Position script. This yields a new template that looks like the following.

```
<template
  id="smResource"
  name="Summary Resource Pick"
  compset="Resource">

  <portal
    id="value"
    style="lblSummary">
    <label
      field="resShort">
      </label>
    </portal>

  <portal
    id="name"
    style="lblSummary">
    <label
      field="name">
      </label>
    </portal>

  <position><![CDATA[
    ~set up our height based on our tallest portal
    height = portal[name].height

    ~if this is a "sizing" calculation, we're done
    if (issizing <> 0) then
      done
      endif

    ~position the value on the left in a limited span and the name
on the right
    portal[value].width = 55
    perform portal[name].alignrel[ltor,value,5]
    ]]></position>
  </template>
```

The final step is to add the new table portal into the layout. We'll place it beneath the existing two tables, so all we need to do is add the portal reference to the layout and then add an appropriate placement statement to the Position script. The resulting layout should look like the following.

```
<layout
  id="smBasics">
  <portalref portal="smAttrib"/>
  <portalref portal="smDerived"/>
  <portalref portal="smStatus"/>

  <position><![CDATA[
    ~position and size the tables to span the full layout
    perform portal[smAttrib].autoplace
    perform portal[smDerived].autoplace[20]
    perform portal[smStatus].autoplace[20]
    ]]></position>

  </layout>
```

We can now reload everything and see the results. The first thing we see is that the list of attributes includes our hidden attribute for super powers. We also see that the list of derived traits include various traits that should be hidden. So we need to add a suitable List tag expression to the two tables that limits the items that are shown. We can refer to the tables used on the Basics tab for how to handle this, at which point we'll determine that the two List tag expressions should be the ones shown below.

```
<list>!Hide.Attribute</list>

<list>!Hide.Trait</list>
```

Looking more closely at the lists of attributes and derived traits, we'll also realize that the information being shown is incorrect. The "trtFinal" field is being shown instead of the proper "trtDisplay" field. So we can fix this within the "value" portal of the "smTrait" template.

We should also provide more space for the display of the various trait values. If a trait is something especially wide like "d12+2", it will be cut off. So we can modify the Position script to double the width of the "value" portal to 40.

At this point, it looks like our summary panel is ready to go.

THE ABILITIES PANEL

The next summary panel is where we'll show all the various abilities for the character. As we did above, we'll adapt the existing "Abilities" panel, which can be found in the file "summ_abilities.dat".

The current panel contains two tables, one for skills and one for abilities. We want to re-use the skills and we'll adapt the abilities table for showing racial abilities. So that leaves us two tables short, as we need additional tables for showing edges and hindrances. This can be readily handled by copying the existing "smAbility" portal twice and revising each into the tables we need. Since edges and hindrances both derive from the "Ability" component, we can also share the same template across all three tables. This results in two new tables like the ones below.

```
<portal
  id="smEdge"
  style="tblInvisSm">
    showtemplate="smAbility"
    scrollable="no">
    </table_fixed>
  </portal>

<portal
  id="smHinder"
  style="tblInvisSm">
  <table_fixed
    component="Hindrance"
    showtemplate="smAbility"
    scrollable="no">
    </table_fixed>
  </portal>
```

We must now add the two table portals into the layout. We'll position the edges at the top, with the hindrances beneath it. Racial abilities and skills be follow. We'll also shrink the spacing between each grouping for this summary panel. This yields a revised layout that looks like the one below.

```
<layout
  id="smAbility">
  <portalref portal="smEdge"/>
  <portalref portal="smHinder"/>
  <portalref portal="smAbility"/>
  <portalref portal="smSkill"/>

  <position><![CDATA[
    ~position and size the tables to span the full layout
    perform portal[smEdge].autoplace
    perform portal[smHinder].autoplace[10]
    perform portal[smAbility].autoplace[10]
    perform portal[smSkill].autoplace[10]
    ]]></position>

  </layout>
```

When we reload everything and take a look at the results, we'll notice that the edges and hindrances are also listed in the table of abilities. Looking more closely at everything, we'll discover that the racial abilities use the "Ability" component, but edges and hindrances also derive from that component. Since the table shows all picks that derive from the "Ability" component, it makes sense that they would be included.

At this point, we have a couple of options regarding how to proceed. The expedient solution is to add a List tag expression to the table portal that omits all edges and hindrances. However, that not necessarily the best solution. Let's consider how racial abilities compare with edges and hindrances. If racial abilities were a more generalized grouping of edges and hindrances, then the current structure would make sense. Unfortunately, racial abilities are actually a completely separate concept from edges and hindrances. So what we really need is a separate component and component set for racial abilities. Then we can have all three different object types share the general mechanisms provided by the "Ability" component.

Open the file "traits.str" and locate the "Hindrance" component. Our new racial abilities component will work very similarly to hindrances, so we'll start by copying the "Hindrance" component. We can now assign a new id like "RaceAbil", and we'll need both the identity tag group and a "SpecialTab" tag. We also need to copy the racial ability to the actor. The other two scripts are specific to hindrances, so we can delete them. This leaves us with a new component the looks like the following.

```
<component
  id="RaceAbil"
  name="Racial Ability"
 autocompset="no">

  <identity group="RaceAbil"/>

  <tag group="SpecialTab" tag="RaceAbil"/>

  <eval index="1" phase="Setup" priority="5000"><![CDATA[
    perform forward[RaceAbil.?]
    ]]></eval>

  </component>
```

Now that the new component is in place, we can define a suitable component set. Racial abilities rely on the common mechanisms of the "Ability" component, the new "RaceAbil" component, and the mechanisms provided by the "SpecialTab" component. This yields a new component set like the one below.

```
<compset
  id="RaceAbil">
  <compref component="RaceAbil"/>
  <compref component="Ability"/>
  <compref component="SpecialTab"/>
  </compset>
```

There is one thing that we're forgetting, though. We've reference a new tag within the "SpecialTab" group, but that tag hasn't been defined yet. Open the file "tags.1st" and locate the "SpecialTab" tag group. Add a new tag for racial abilities, which should look like the example below.

```
<value id="RaceAbil" name="Racial Ability" order="60"/>
```

Since we've defined a new component set for racial abilities, we now need to go through all our racial ability things and change them appropriately. All of them are defined in the file "thing_races.dat", so open that file. We can do a global search and replace operation on the file to change all things with the component set "Ability" to "RaceAbil". Once that's done, everything should compile again.

The final step is to return to the table within the summary panel. For the "smAbility" table portal we created earlier, change the component to "RaceAbil". This will limit the table to only show racial abilities.

Reload the files and assess how everything is behaving. Unless there was a mistake somewhere, it should all be working as we intended now.

THE AMORY PANEL

Our third and final summary panel is for showing weapons and armor possessed by the character. Fortunately, we don't need to do much here. All of the handling for armor, melee weapons, and ranged weapons is already in place and working nicely. So all we need to do is add a table of special weapons at the bottom.

As we've done previously, we can copy an existing table portal and adapt it easily. This yields the new portal below.

```
<portal
  id="smSpecial"
  style="tblInvisSm">
  <table_fixed
    showtemplate="smWeapon"
    scrollable="no">
    </table_fixed>
  </portal>
```

All that's left is to add the table portal to the layout. We'll just append it at the end, resulting in the revised layout shown below.

```
<layout
  id="smArmory">
  <portalref portal="smDefense"/>
  <portalref portal="smRange"/>
  <portalref portal="smMelee"/>
  <portalref portal="smSpecial"/>

  <position><![CDATA[
    ~position and size the tables to span the full layout
    perform portal[smDefense].autoplace
    perform portal[smRange].autoplace[20]
    perform portal[smMelee].autoplace[20]
    perform portal[smSpecial].autoplace[20]
    ]]></position>

  </layout>
```

That's all there is to it. We've now got our final summary panel working as it should.

CHARACTER OUTPUT

CHARACTER SHEET DESIGN (SAVAGE)

It's finally time to shift our focus to a character sheet. With HL, you are not limited to a single character sheet. You can create various designs, with a one-page sheet that is highly compact, another that is a detailed portfolio, and yet another that is a two-page balance of the previous two (that users can print out two-sided). For the purposes of the walk-through, we're only going to create a single character sheet, but the process will be basically the same for any character sheet you choose to create.

CHOOSING THE LAYOUT

There are two important factors you need to consider when designing the layout of your character sheet. First, most game systems have a "standard" character sheet that comes in the rulebook or is made available on the publisher's website. Providing a character sheet that is similar to the standard format is generally a good thing, since it provides the user with a familiar reference. The user instinctively knows where to look for information, since it's in the same place he's used to finding it.

The second factor to consider is that the standard character sheet provided by the publisher is designed for use throughout the life a character and that will work for all types of characters. As such, it includes lots of space that will be unused for various character types, and it also includes information that will actually make the character sheet more distracting for the user.

Let's look at Savage Worlds for some examples. The character sheet in the back of the book has sections for hindrances, edges, skills, and powers. However, if you create a character with no powers, that powers section is irrelevant. Same for hindrances. If you create a character that emphasizes skills instead of edges, you may run out of space for skills, and all the space for edges is wasted.

Each attribute and skill entry on the character sheet shows all of the different die types. This makes it easy for a player to mark the

appropriate die type for each. However, your data files already know the proper die type to use and can show just that die type. Showing all the different die types provides no useful information to the user and clutters up the character sheet with additional "noise".

Clearly, the second factor is directly contrary to the first one here. Your task is to find an appropriate balance between these two competing priorities. There is no "right" answer, so you'll just have to use your best judgement.

There is a third factor that you should also keep in mind. Since HL has all of the calculated information for the character, you can readily include those calculated results on the character sheet. You can also include details about abilities, weapons, etc. Many character sheets provided by publishers only provide a place to enter the names of items, with little room for the details. So there may be times when it's more useful to include information that the standard character sheet omits. Since you obviously play the game, think of the information you want to have available to you. Also, remember back to the first time you played the game, since new players often find it helpful to see information that an expert has already memorized.

In the case of Savage Worlds, the design task become even more complex. While there is a single character sheet in the back of the rulebook, there are a dozen more character sheets provided on the publisher's website. Each of these sheets provides all the core material, but there are a wide variety of layouts used. As such, there really isn't a "standard" sheet that we can use as a guideline. We'll just have to figure something out that seems to work well.

Designing the Sheet

The first two things we need to decide are the number of pages our character sheet will contain and whether we'll be using a portrait or landscape orientation for our sheets. All of the character sheets provided by the publisher are a single page, but they also preclude the inclusion of lots of helpful information that can be useful during game play due to the compact format.

At this point, it's generally best to pull out a few blank sheets of paper and start sketching until you settle on what you think is best. You can draw rectangles where various tables of information will appear on the sheet to give yourself a good idea of how everything will be laid out. Be sure to try a few different ideas, just to be sure that the one you pick is your favorite.

When sketching, keep one important detail in mind. Character sheets in HL are adaptive, which means that tables will size themselves based on the information that needs to be output. The best way to leverage this adaptive behavior is to have your character sheet flow in columns. That way, if the information in one table is short, the next table will follow immediately below it. If a table is long, then the following table may get pushed into the next column. The use of two or three columns is best for portrait orientation, while three or four columns are best for landscape orientation.

Another factor to keep in mind is how much time and effort you want to invest in implementing your character sheet. The Skeleton files provide a framework for a character sheet that will often work reasonably well for most game systems. The provided sheet strives to put all the information on a single page in a two-column format using portrait orientation. If not all the information fits on the first page, a second "spillover" page is output that contains whatever information remains to be output. You can see how this looks by loading the Sample game system, creating a character, and then previewing the character sheet.

One option to consider is using the provided framework for an initial character sheet that you can develop quickly, after which you can go back and create a second character sheet that is more in line with what you think is ideal. Given that this is our first character sheet, we'll adopt this approach. We can then sketch out our design on a sheet of blank paper and get the implementation underway.

Reviewing The Design Sketch

Once our sketch is complete, we can take a final look at it to see if there are any problems. We'll take this opportunity to walk through the design here before we start implementing the changes.

Our goal is to fit everything onto a single page, just like all the character sheets provided on the publisher's website, and we'll use portrait orientation for the sheet. Since the Kit makes it relatively easy, we'll also provide a second spill-over sheet, just in case a particular character is too complex to fit on one page. We'll use the two-column format that is already provided for us by the Skeleton files in the interest of getting something working quickly.

Down the left column, we'll show the following sections, in this sequence: basic information, attributes, derived trait, skills, racial abilities, hindrances, and edges. In some case, not all the edges will fit, so we'll wrap those to the top of the second column, if necessary. The second column will contain powers, weapons, armor, gear, injuries, and a place to track the character's condition and power points.

Since we're using a two-column format, there are quite a few sections where we won't be able to use the space very efficiently. For example, attributes, derived traits, skills, and gear definitely don't need a full half-page of width. Consequently, we're going to use a two-column table for those sections, which ought to work nicely. The gotcha with this approach is with skills, since we also need to handle Knowledge skills, for which the domain needs to be shown. That won't fit within a two-column table. So we'll have a separate table of Knowledge skills that uses one column beneath the regular skills in a two-column table. That should keep things both compact and clear.

The other odd cases are the injuries table and character condition tracking. The injuries table can be short and sweet, and the condition tracking can be squeezed into various places. So we'll use a narrow table for injuries at the bottom of the right column and put the condition tracking details to the right of the injuries.

We've got a basic plan mapped out, so we're now ready to start implementing our character sheet.

Character Sheet Design (Savage)

It's finally time to shift our focus to a character sheet. With HL, you are not limited to a single character sheet. You can create various designs, with a one-page sheet that is highly compact, another that is a detailed portfolio, and yet another that is a two-page balance of the previous two (that users can print out two-sided). For the purposes of the walk-through, we're only going to create a single character sheet, but the process will be basically the same for any character sheet you choose to create.

Choosing the Layout

There are two important factors you need to consider when designing the layout of your character sheet. First, most game

systems have a "standard" character sheet that comes in the rulebook or is made available on the publisher's website. Providing a character sheet that is similar to the standard format is generally a good thing, since it provides the user with a familiar reference. The user instinctively knows where to look for information, since it's in the same place he's used to finding it.

The second factor to consider is that the standard character sheet provided by the publisher is designed for use throughout the life a character and that will work for all types of characters. As such, it includes lots of space that will be unused for various character types, and it also includes information that will actually make the character sheet more distracting for the user.

Let's look at Savage Worlds for some examples. The character sheet in the back of the book has sections for hindrances, edges, skills, and powers. However, if you create a character with no powers, that powers section is irrelevant. Same for hindrances. If you create a character that emphasizes skills instead of edges, you may run out of space for skills, and all the space for edges is wasted.

Each attribute and skill entry on the character sheet shows all of the different die types. This makes it easy for a player to mark the appropriate die type for each. However, your data files already know the proper die type to use and can show just that die type. Showing all the different die types provides no useful information to the user and clutters up the character sheet with additional "noise".

Clearly, the second factor is directly contrary to the first one here. Your task is to find an appropriate balance between these two competing priorities. There is no "right" answer, so you'll just have to use your best judgement.

There is a third factor that you should also keep in mind. Since HL has all of the calculated information for the character, you can readily include those calculated results on the character sheet. You can also include details about abilities, weapons, etc. Many character sheets provided by publishers only provide a place to enter the names of items, with little room for the details. So there may be times when it's more useful to include information that the standard character sheet omits. Since you obviously play the game, think of the information you want to have available to you. Also, remember back to the first time you played the game, since new players often find it helpful to see information that an expert has already memorized.

In the case of Savage Worlds, the design task become even more complex. While there is a single character sheet in the back of the rulebook, there are a dozen more character sheets provided on the publisher's website. Each of these sheets provides all the core material, but there are a wide variety of layouts used. As such, there really isn't a "standard" sheet that we can use as a guideline. We'll just have to figure something out that seems to work well.

Designing the Sheet

The first two things we need to decide are the number of pages our character sheet will contain and whether we'll be using a portrait or landscape orientation for our sheets. All of the character sheets provided by the publisher are a single page, but they also preclude the inclusion of lots of helpful information that can be useful during game play due to the compact format.

At this point, it's generally best to pull out a few blank sheets of paper and start sketching until you settle on what you think is best. You can draw rectangles where various tables of information will appear on the sheet to give yourself a good idea of how everything will be laid out. Be sure to try a few different ideas, just to be sure that the one you pick is your favorite.

When sketching, keep one important detail in mind. Character sheets in HL are adaptive, which means that tables will size themselves based on the information that needs to be output. The best way to leverage this adaptive behavior is to have your character sheet flow in columns. That way, if the information in one table is short, the next table will follow immediately below it. If a table is long, then the following table may get pushed into the next column. The use of two or three columns is best for portrait orientation, while three or four columns are best for landscape orientation.

Another factor to keep in mind is how much time and effort you want to invest in implementing your character sheet. The Skeleton files provide a framework for a character sheet that will often work reasonably well for most game systems. The provided sheet strives to put all the information on a single page in a two-column format using portrait orientation. If not all the information fits on the first page, a second "spillover" page is output that contains whatever information remains to be output. You can see how this looks by loading the Sample game system, creating a character, and then previewing the character sheet.

One option to consider is using the provided framework for an initial character sheet that you can develop quickly, after which you can go back and create a second character sheet that is more in line with what you think is ideal. Given that this is our first character sheet, we'll adopt this approach. We can then sketch out our design on a sheet of blank paper and get the implementation underway.

Reviewing The Design Sketch

Once our sketch is complete, we can take a final look at it to see if there are any problems. We'll take this opportunity to walk through the design here before we start implementing the changes.

Our goal is to fit everything onto a single page, just like all the character sheets provided on the publisher's website, and we'll use portrait orientation for the sheet. Since the Kit makes it relatively easy, we'll also provide a second spill-over sheet, just in case a particular character is too complex to fit on one page. We'll use the two-column format that is already provided for us by the Skeleton files in the interest of getting something working quickly.

Down the left column, we'll show the following sections, in this sequence: basic information, attributes, derived trait, skills, racial abilities, hindrances, and edges. In some case, not all the edges will fit, so we'll wrap those to the top of the second column, if necessary. The second column will contain powers, weapons, armor, gear, injuries, and a place to track the character's condition and power points.

Since we're using a two-column format, there are quite a few sections where we won't be able to use the space very efficiently. For example, attributes, derived traits, skills, and gear definitely don't need a full half-page of width. Consequently, we're going to use a two-column table for those sections, which ought to work nicely. The gotcha with this approach is with skills, since we also need to handle Knowledge skills, for which the domain needs to be shown. That won't fit within a two-column table. So we'll have a separate table of Knowledge skills that uses one column beneath the regular skills in a two-column table. That should keep things both compact and clear.

The other odd cases are the injuries table and character condition tracking. The injuries table can be short and sweet, and the condition tracking can be squeezed into various places. So we'll use a narrow table for injuries at the bottom of the right column and put the condition tracking details to the right of the injuries.

We've got a basic plan mapped out, so we're now ready to start implementing our character sheet.

Implementing the Character Sheet (Savage)

Our design is in place, so it's time to start actually implementing our new character sheet. Since we're adapting the sheet provided by the Skeleton files, we'll find what we need in the file "sheet_standard1.dat".

Before We Get Started

We're going to be using this sheet both as our starting point for our changes and as a source of examples for how to do various tasks. As such, we should make a copy of this file somewhere (not within our data file folder). We can keep the copy around for use as a reference or guide on how to do things. We can then freely change the file in our game system folder to morph it into our new character sheet for Savage Worlds.

During the development of our character sheet, we're going to be taking a look at how things behave dozens of times. There is generally no need to print anything during our testing. Instead, you can use the Print Preview mechanism within HL, which can be accessed quickly via the keystroke sequence <Alt+F><V>. After doing a quick-reload, you'll find yourself following with these keystrokes on a regular basis.

We also want to see a meaningful character being displayed when testing our character sheet. If you haven't done so already, create a character for testing. It's probably best to have the character be invalid and have more capabilities than a starting character normally should. This way, you really see how all the different facets of the character sheet will work. For example, give the character an arcane background and powers to see how that stuff works. Give him an assortment of skills, hindrances, edges, weapons, armor, and gears. That way, when you're testing things, you'll get a good sense of how everything looks by just using the existing character. The entire process goes much more quickly this way.

Character Basics

The provided character sheet that we using as our template places the basic information in the top left corner. We'll start there for simplicity and then develop our character sheet in a systematic fashion by proceeding down the left column and then down the right column.

The character name is placed within the header bar for the section, so that's a nice prominent place that works well for us. Beneath that, the Skeleton files have three visible sections of information, with each being less prominent than the one above. The first shows the player name in a large bold font. The official character sheets don't include this and it takes up valuable space when we're trying to keep things compact, so we'll just omit that from our sheet. The second section shows the character points in bold, which don't apply to Savage Worlds, and the third shows basic character details. Having two sections ought to work nicely for us, with one in bold and one not, but we need to re-apportion the information and add some details.

In the first section, we should show the two most important facets of the character that will be included in the character basics. Those are the character's race and rank (with XP). If we do this, that leaves the physical characteristics for display beneath. We could include the hair, eyes, and skin tone in this section, but that would generally require a second line of text on the character sheet and we're trying to keep everything compact. So we'll leave that information out and switch to centering the remaining details on the line.

All of this information is managed via the "oHeroInfo" portal, so we'll adapt it for our purposes. The text is simply synthesized by a single script. Once we finish revising it, the result should look similar to the new script below.

```
~start with the character's race
 var temp as string
 temp = hero.firstchild["component.Race"].field[name].text
 if (empty(temp) <> 0) then
   temp = "Unknown"
   endif  @text &= "{size 36}Race: " & temp

~append the rank and XP, showing any unspent advances
 var rankvalue as number
 var ranktext as string
 rankvalue = herofield[acRank].value
 call RankName
 @text &= "; " & ranktext & "  ("
 @text &= hero.child[resXP].field[resMax].text & " XP)"
 if (#resleft[resAdvance] > 0) then
   @text &= "; " & #resleft[resAdvance] & " Advance(s)"
   endif

~sart a new line and turn off bold
 @text &= "{br}{/b}"

~append the gender, age, height, and weight
 if (hero.child[mscPerson].field[perGender].value = 0) then
   @text &= "Male"
 else
   @text &= "Female"
   endif
 @text &= "; Age: " & hero.child[mscPerson].field[perAge].text
 @text &= "; Height: " &
hero.child[mscPerson].field[perHeight].text
 @text &= "; Weight: " &
hero.child[mscPerson].field[perWeight].text
```

Attributes

Moving downward, the next section we want to include is the attributes, and they conveniently happen to be next in the provided files. However, the way attributes are handled in the Skeleton files is a bit different from how we're going to be handling them. In the Skeleton files, attributes have a header across the top and a bunch of pieces for display. For our purposes, all we need is a basic title at the top and relatively simple display elements for each attribute. So we've got a bit of work to do in revamping how attributes work.

The first thing we need to do is revise the attributes table portal appropriately, which has the id "oAttribute". We'll eliminate the use of a header template and designate that we'll be using a two-column table. We then need to add our simple header by adding a suitable "headertitle" element. Lastly, we need to accommodate the hidden attribute for super powers appropriately, which requires we define a List tag expression that omits hidden attributes. This results in a revised portal that looks similar to the one below.

```
<portal
```

```
    id="oAttribute"
    style="outNormal">
    <output_table
      component="Attribute"
      showtemplate="oAttrPick"
      showsortset="explicit"
      columns="2">
      <list>!Hide.Attribute</list>
      <headertitle><![CDATA[
        @text = "Attributes"
        ]]></headertitle>
      </output_table>
    </portal>
```

We next shift our focus to the "oAttrPick" template used for displaying each attribute. We'll start by changing the vertical margin to zero, and we can increase it later if want. The next step is to eliminate all of the existing portals that are used as the header, since we won't be using them. We can also eliminate the Header script that is no longer needed. This leaves us with six portals that we need to deal with. We obviously want to keep the name portal, and we'll also need portals for both the final die-type and any adjustment that applies (just like we did on the Basics tab). So we can delete all the portals we don't need and add new ones for what we do need.

We can look at the approach we used on the Basics panel for guidance on how to define our two new portals. The portal showing the adjustment can be essentially copied, with the caveat that we need to convert it over to be an output portal with a corresponding output style. We could do the same for the attribute value itself, except that we'd end up with very poor results. The problem is that the fields that are automatically generated for our traits to contain display bitmaps are tailored for on-screen display. So we get colored bitmaps that are going to be very poor quality on printers that are comparatively very high resolution.

We need to come up with a different approach for displaying the attribute die-type bitmaps. Fortunately, the Skeleton files include a separate assortment of bitmaps for the various die-types that are specifically intended for use within character sheet output. We'll need to identify the proper bitmap to use in each case, just like we did within the "FinalRoll" procedure that we defined earlier. In fact, we might as well implement this logic in a new procedure as well, since we're going to need the same logic when we display skills.

Open the file "procedures.dat" and locate the existing "FinalRoll" procedure. We need a similar procedure that synthesizes the proper die-type bitmap for printed output. The Kit provides two sets of bitmaps for printed die-type output, one for showing an "active" die and the other for showing an "inactive" die. This makes it possible to show a sequence of die-type bitmaps, with the proper one shown prominently as "active" and the others shown more faintly as "inactive". In the interest of keeping things compact, we won't be using that approach for our character sheet, but you can easily do it for your own sheet if you want. All we'll be using is the proper "active" bitmap, which results in the new procedure shown below.

```
  <procedure id="OutputDie" scripttype="none"><![CDATA[
    ~declare variables that are used to communicate with our
caller
    var dietype as number
    var dietext as string

   ~bound our final die type appropriately
    var final as number
    final = dietype
    final = 2
   elseif (final > 6) then
    final = 6
    endif

   ~convert the final value for the trait to the proper die type
bitmap for output
    ~Note: Be sure to scale the bitmap appropriately for the size
we want.
    final *= 2
    dietext = "{bmpscale 9 d" & final & "_outputactive}"
    ]]></procedure>
```

Now that we've got our procedure in place, we can easily define our portals and tailor the Position script accordingly. This yields a revised template that looks like the following.

```
 <template
    id="oAttrPick"
    name="Output Attributes Table"
    compset="Attribute"
    marginvert="0">

   <portal
     id="name"
     style="outNameLg">
     <output_label
       field="name">
       </output_label>
     </portal>

   <portal
     id="value"
     style="outValueLg">
     <output_label>
       <labeltext><![CDATA[
         var dietype as number
         var dietext as string
         dietype = field[trtFinal].value
         call OutputDie
         @text = dietext
         ]]></labeltext>
       </output_label>
     </portal>

   <portal
     id="adjust"
     style="outNameLg">
     <output_label>
       <labeltext><![CDATA[
         if (field[trtNetRoll].value = 0) then
           @text = ""
         else
           @text = signed(field[trtNetRoll].value)
           endif
         ]]></labeltext>
       </output_label>
     </portal>

   <position><![CDATA[
     ~our height is driven by the tallest portal
     height = portal[value].height
     if (issizing <> 0) then
       done
       endif

    ~center everything vertically within the template
     perform portal[name].centervert
     perform portal[value].centervert
     perform portal[adjust].centervert

    ~position everything horizontally
     portal[name].left = 65
```

```
    portal[value].left = 340
    perform portal[adjust].alignrel[ltor,value,5]
    ]]></position>

  </template>
```

Reload the data files and take a look at our character sheet. It looks reasonable, but the empty gap between the attribute names and the die-type bitmaps makes some of the pieces seem rather isolated from one another, especially when the attribute name is short. Visually associating the attribute name and value would be an immense help in making our character sheet look attractive. Fortunately, the Kit provides a convenient way of accomplishing this objective.

There is a special output portal that is specifically designed for use in situations like this ("output_dots"). It places a series of dots between two portals, and you can see it in use within the derived traits, weapons, and skills within the provided character sheet. We're going to use this technique for displaying our attributes. Doing so requires that we add a new portal for the dots and then position and size it appropriately. At the bottom of the list of portals, add the new portal shown below. In addition, add the lines of script code at the end of the Position script to integrate the portal properly.

```
<portal
  id="dots"
  style="outDots">
  <output_dots>
  </output_dots>
</portal>
```

```
 ~extend the dots from the right of the name across to the value
on the right
  perform portal[dots].alignrel[ltor,name,0]
  portal[dots].width = portal[value].left - 5 - portal[dots].left
  perform portal[dots].centervert
```

Reload everything and take a look at the character sheet. This looks much better. In fact, we'll use this technique in additional tables below, which will both visually associate elements better and give our character sheet a consistent look.

HEALTH AND POWER

Within the provided Skeleton files, this is a section where you can include important resources that need to be tracked, such as health. Savage Worlds traditionally uses a different approach to these resources, so we don't need this section in the character. We'll add different handling later when we reach that point in our development.

Deleting the section from the character sheet entails a few steps, since there are multiple pieces involved.

1. First of all, this section is handled via the "oPower" table portal, so we can start by deleting that portal.
2. The portal referenced the "oPowerPick" template, so we next delete that template.
3. The portal is utilized within the "oLeftSide" layout, so we need to delete it from there. This includes removing the "portalref" element and the line of code in the Position script that places the portal.

That should cover it. Reload the file and preview the character sheet. The section is now eliminated.

DERIVED TRAITS

Moving down the left column, the next section is the derived traits. The Skeleton files provide this section already, and all we need to do is adapt it appropriately. The table portal has the id "oDerived", so we'll start there. We want to use the standard style for the table instead of the grid-based one used in the Skeleton files, so we first change the style to "outNormal". Then we can change the component to the more restrictive "Derived" and switch to a two-column table. The final change is to the List tag expression, which needs to omit any traits that are specifically hidden. This yields a table portal like the one shown below.

```
<portal
  id="oDerived"
  style="outNormal">
  <output_table
    component="Derived"
    showtemplate="oDerivPick"
    showsortset="explicit"
    columns="2">
    <list>!Hide.Trait</list>
    <headertitle><![CDATA[
      @text = "Derived Traits"
    ]]></headertitle>
  </output_table>
</portal>
```

We now shift our focus to the template, which has the id "oDerivPick". We'll start with margins that are designed for a single-column table that needs to leave room for the grid around each item. We'll eliminate the horizontal margin, since we're packing the derived traits into two columns. We'll also reduce the vertical margin to zero for now, leaving ourselves the option to increase it again later.

The existing template simply shows a name and value, with dots between. For Savage Worlds, we need this same behavior, plus we also need to provide an empty box next to each value where the user can note adjustments to the value during play. For this, we'll add a new portal that contains just a blank space (we have to specify something, else the compiler will complain). We'll use the "outGreyBox" style to put a box around the portal, thereby providing a place where the user can note adjustments.

Instead of positioning the portals at the left and right edges, we'll just align them from left to right. This yields the following revised template.

```
<template
  id="oDerivPick"
  name="Output Derived Traits Table"
  compset="Trait"
  marginvert="8">

  <portal
    id="name"
    style="outNameLg">
    <output_label
      field="name">
    </output_label>
  </portal>

  <portal
    id="value"
    style="outValueLg">
```

```
      <output_label
        field="trtDisplay">
        </output_label>
      </portal>

    <portal
      id="adjust"
      style="outGreyBox">
      <output_label
        text=" ">
        </output_label>
      </portal>

    <portal
      id="dots"
      style="outDots">
      <output_dots>
        </output_dots>
      </portal>

    <position><![CDATA[
      ~our height is based on the tallest portal (they should all be
the same)
      height = portal[name].height
      if (issizing <> 0) then
        done
        endif

      ~setup appropriate widths for the various portals
      portal[adjust].width = 100

      ~center everything vertically
      perform portal[name].centervert
      perform portal[value].centervert
      perform portal[adjust].centervert
      perform portal[dots].centervert

      ~position everything horizontally
      portal[name].left = 45
      portal[value].left = 355
      perform portal[adjust].alignrel[ltor,value,30]

      ~extend the dots from the right of the name across to the
value on the right
      perform portal[dots].alignrel[ltor,name,0]
      portal[dots].width = portal[value].left - 5 - portal[dots].left
      ]]></position>

    </template>
```

After reloading the files and taking a look at the character sheet, this looks pretty good, except for one glaring problem. The boxes for the user to write in adjustments are touching, so we need to insert a bit of spacing. We can go back to the vertical margins and increase them a bit. If we increase the margin to a value of 5, everything looks great.

CHARACTER SHEET PHASE 2 (SAVAGE)

Our character sheet is starting to come together. We've got some of the most basic sections in place and we've got some momentum going, so we'll keep on rolling.

SWAPPING SECTIONS AROUND

At this point, we need to make some structural adjustments to the sheet provided with the Skeleton files. We want to use the skills and special abilities sections, but they are currently shown in the righthand column. Similarly, armor and weapons are on the left, but we want to show them on the right. So the next thing we'll do is move these sections around, after which we can resume editing the individual sections.

Armor and weapons are both encapsulated by the "oArmory" layout, so we can move both of those at the same time by moving the layout. There are some interesting wrinkles associated with this layout that we'll need to deal with at some point, but we don't need to worry about them now. The "oArmory" layout is managed directly by the sheet, so we can move it to the right column within the Position script of the sheet. We'll have to fix this later, but for now we'll just move it to the bottom on the right side of the sheet. This is accomplished by moving the code that positions the layout to the end of the script and revising it properly for being at the bottom on the right. The new code should look like below.

```
  ~position the armory layout within the remaining space on the
right
  layout[oArmory].width = colwidth
  layout[oArmory].top = layout[oRightSide].bottom +
scenevalue[sectiongap]
  layout[oArmory].left = layout[oLogos].left
  layout[oArmory].height = extent - scenevalue[sectiongap] -
layout[oArmory].top
  perform layout[oArmory].render
```

The next thing we need to do is move the skills and abilities sections from the right side to the left side. Fortunately, doing this is extremely easy. The skills and abilities are both handled via separate table portals. Both of these portals are currently managed within the "oRightSide" layout and positioned via the "autoplace" mechanism. Consequently, all we need to do is move them into the "oLeftSide" layout and place them properly within the new Position script.

Within the "oRightSide" layout, delete the two "portalref" elements for the "oAbility" and "oSkills" portals. Now go to the "oLeftSide" layout and add the elements there. Return to the "oRightSide" layout and delete the two "autoplace" operations on the portals we just removed. Then go to the "oLeftSide" layout and insert the two lines of code into the Position script there. When you're done, the two revised layouts should look like the below.

```
 <layout
   id="oLeftSide">
   <portalref portal="oHeroName"/>
   <portalref portal="oHeroInfo"/>
   <portalref portal="oAttribute"/>
   <portalref portal="oDerived"/>
   <portalref portal="oSkills"/>
   <portalref portal="oAbility"/>
   <position><![CDATA[
    ~position the hero name at the top with the hero details
beneath the name
     perform portal[oHeroName].autoplace[0]
     perform portal[oHeroInfo].autoplace[15]

    ~position the tables next
     perform portal[oAttribute].autoplace
     perform portal[oDerived].autoplace
     perform portal[oSkills].autoplace
     perform portal[oAbility].autoplace

    ~our layout height is the extent of the elements within
     height = autotop
     ]]></position>
   </layout>
```

```
 <layout
   id="oRightSide">
   <portalref portal="oGear"/>
```

```
    <templateref template="oPortrait" thing="actor"/>
    <position><![CDATA[
     ~position the character portrait at the top and the various
tables beneath
     perform template[oPortrait].autoplace
     perform portal[oGear].autoplace

     ~our layout height is the extent of the elements within
     height = autotop
     ]]></position>
    </layout>
```

That's all there is to it. Reload the data files and preview the character. Everything has now been swapped the way we intended.

SKILLS

Now that the swap is completed, we can resume with the next section on the left side, which happens to be skills. We'll handle skills the same way that we did attributes, using a two-column table and showing the die-type and adjustment for each skill. However, there is an important wrinkle with skills. We have to handle the knowledge skills specially, since we need to show the domain for those skills, and that means we can't use a two-column format for those skills.

The simplest solution is to create a second table portal that is the same as the standard portal for skills, with the key difference being a one-column format. Obviously, we also need to use a different List tag expression in each, with one table listing the non-knowledge skills and the other listing only the knowledge skills. Just to be safe, we should anticipate that a supplement may introduce new skills that require domains, and we need to handle those smoothly as well. So we'll differentiate based on whether the skill uses a domain or not.

We'll start by modifying the existing "oSkills" portal to properly display all skills in the two-column format. We can copy the portal for use with domain-based skills and give it the id "oSkillsDom". This portal then specifies a single column. Now we need to figure out how to differentiate between the domain-based skills and non-domain skills. Fortunately, this is easy. When we setup the handling of domains for skills, we based it on the presence of the "User.NeedDomain" tag, so we can do the same now within the List tag expression. This yields the following two table portals.

```
<portal
  id="oSkills"
  style="outNormal">
  <output_table
    component="Skill"
    showtemplate="oSkillPick"
    columns="2">
    <list>!User.NeedDomain</list>
    <headertitle><![CDATA[
      @text = "Skills"
     ]]></headertitle>
  </output_table>
</portal>

<portal
  id="oSkillsDom"
  style="outNormal">
  <output_table
    component="Skill"
    showtemplate="oSkillPick">
    <list>User.NeedDomain</list>
  </output_table>
</portal>
```

Now we can look at the "oSkillPick" template that is used to display each skill. Since we want our skills to look and behave basically the same as how we've already got attributes working, it's actually easiest to delete the current template, copy the attribute template, and adapt the attributes template slightly for skills. So that's how we'll proceed.

Once the attribute template is cloned, we can revise it. First, we need to give it the appropriate id of "oSkillPick". The four portals are the same for skills, so no changes are needed there. However, skills possess domains, so we need to handle that properly within our spacing. Within the Position script, we need to change the horizontal positions assigned to the "name" and "value" portals. Since the template is being used in both of the table portals, we actually want different positions based on whether the skill has a domain or not. Putting this all together results in the revised Position script shown below.

```
~our height is driven by the tallest portal
 height = portal[value].height
 if (issizing <> 0) then
   done
   endif

~position everything horizontally
 if (tagis[User.NeedDomain] = 0) then
   portal[name].left = 40
   portal[value].left = 405
 else
   portal[name].left = 105
   portal[value].left = 900
   endif
 perform portal[adjust].alignrel[ltor,value,5]

~limit the name to the value portal edge
 if (portal[name].right > portal[value].left - 10) then
   portal[name].width = portal[value].left - portal[name].left – 10
   endif

~size the name to fit the available space and limit to one line
 perform portal[name].sizetofit[36]  portal[name].lineheight = 1

~extend the dots from the right of the name across to the value
on the right
 perform portal[dots].alignrel[ltor,name,0]
 portal[dots].width = portal[value].left - 5 - portal[dots].left

~center everything vertically within the template
 perform portal[name].centervert
 perform portal[value].centervert
 perform portal[adjust].centervert
 perform portal[dots].centervert
```

If we reload the data files and preview the character again, we should now see our skills in a proper two-column layout. Once we add a Knowledge skill to the character and assign it a domain, we can preview it again. But our Knowledge skill isn't appearing. It seems we forgot to do something.

The problem is that we haven't integrated the new table portal for domain-based skills into the layout yet. So we'll do that now. Locate the "oLeftSide" layout and add a new "portalref" element for the "oSkillsDom" portal. Then add an "autoplace" statement to the Position script, immediately after the placement of the "oSkills" portal. The reload and preview again.

Now our Knowledge skill is showing up, but there is a gap between the normal skills and the domain-based skills. This is because the

"autoplace" logic is using a default gap that is setup by the sheet. We don't want a gap between two skills tables, so we can explicitly use a gap of zero by using a statement like the one below.

```
perform portal[oSkillsDom].autoplace[0]
```

It's time for one last reload and preview. This time, everything is looking good for skills.

Racial Abilities

Next on the list is racial abilities. The Skeleton files already provide a table portal and template for showing abilities, and we'll simply adapt it for display of racial abilities. For the portal itself, we need to change to the more restrictive "RaceAbil" component so that we only show racial abilities. We also must change the header to specify "Racial" instead of "Special". This results in the following portal.

```
<portal
  id="oAbility"
  style="outNormal">
  <output_table
    component="RaceAbil"
    showtemplate="oAbilPick">
    <headertitle><![CDATA[
      @text = "Racial Abilities"
     ]]></headertitle>
  </output_table>
</portal>
```

Looking at the template, it already provides exactly what we need. The name of the ability is prominent, and a summary of the ability is shown in the remaining space. In the interest of consistency, we'll use this same template for edges and hindrances, so most characters will generally have lots of abilities. Based on that, the only thing we should probably change is the font size used for the ability name. By shrinking it a little bit, we'll reduce the vertical height of each entry in the table and also give ourselves a little more horizontal space for the summary text.

All we need to do is switch to a suitable style with a slightly smaller font size. The "ofntmedium" font should do nicely. Unfortunately, we also need a style that is left-aligned, and the only style using the medium font is center-aligned. So we can create a new output style that is left-aligned and uses the medium font, as shown below.

```
<style
  id="outMedLt">
  <style_output
    textcolor="000000"
    font="ofntmedium"
    alignment="left">
  </style_output>
</style>
```

Once we've got our new style defined, we can simply change the style associated with the "details" portal to use the "outMedLt" style. Then we can reload the data files and preview again. The racial abilities look great now.

Hindrances and Edges

Both hindrances and edges are grouped together because they are extremely simple and work virtually the same. Both of these facets of the character are built upon the "Ability" component. Since all we're showing is the name and summary on the character sheet, we can readily re-use the same template that we setup for showing racial abilities.

This means that all we need to do is create two new table portals, both of which can be cloned and adapted from the portal used for racial abilities. We simply need to change the unique id, the component being filtered, and the header text. This results in the two new portals below.

```
<portal
  id="oHindrance"
  style="outNormal">
  <output_table
    component="Hindrance"
    showtemplate="oAbilPick">
    <headertitle><![CDATA[
      @text = "Hindrances"
     ]]></headertitle>
  </output_table>
</portal>

<portal
  id="oEdge"
  style="outNormal">
  <output_table
    component="Edge"
    showtemplate="oAbilPick">
    <headertitle><![CDATA[
      @text = "Edges"
     ]]></headertitle>
  </output_table>
</portal>
```

Once the two portals are defined, we then need to integrate them into the "oLeftSide" layout. This entails adding "portalref" elements for each and then positioning them beneath the racial abilities. The revised layout is shown below.

```
<layout
  id="oLeftSide">
  <portalref portal="oHeroName"/>
  <portalref portal="oHeroInfo"/>
  <portalref portal="oAttribute"/>
  <portalref portal="oDerived"/>
  <portalref portal="oSkills"/>
  <portalref portal="oSkillsDom"/>
  <portalref portal="oAbility"/>
  <portalref portal="oHindrance"/>
  <portalref portal="oEdge"/>
  <position><![CDATA[
    ~position the hero name at the top with the hero details
beneath the name
    perform portal[oHeroName].autoplace[0]
    perform portal[oHeroInfo].autoplace[15]

    ~position the tables next
    perform portal[oAttribute].autoplace
    perform portal[oDerived].autoplace
    perform portal[oSkills].autoplace
    perform portal[oSkillsDom].autoplace[0]
    perform portal[oAbility].autoplace
    perform portal[oHindrance].autoplace
    perform portal[oEdge].autoplace

    ~our layout height is the extent of the elements within
    height = autotop
   ]]></position>
</layout>
```

If we reload the data files and preview the character sheet, we'll now see the new sections for hindrances and edges. At this point, the left side of our character sheet is complete.

Shorter Names for Output

At this point, you'll probably have noticed that some of the various edges, hindrances, and racial abilities have reasonably long names. The longer names severely limit the amount of description text that can be included in the character sheet. It would be ideal to utilize shorter names for some of these items.

The Kit makes it easy to add support for shorter names to a group of items in situations like this. If you identify a component where short names would be beneficial, you can set the "hasshortnames" attribute on the component to "yes". Doing so ensures that every thing derived from the component possesses a field with the id of "shortname". You can then specify this field within the definition of things derived from the component. If a particular thing has no "shortname" field, then the field automatically defaults to the contents of the basic name for the thing.

We use the same template for edges, hindrances, and racial abilities. As such, we need to add support for short names to all three. Conveniently, all three share the "Ability" component, so we only need to assign the new behavior to this one component.

Once that's done, we can go back through the various things and define "fieldval" elements to properly assign a "shortname" to those that require one. For example, the "Arcane Background: Super Powers" edge is extremely long and can be readily shorted to "Arcane Super Powers", or something along those lines.

With short names in place, our last step is to make use of the new field. This simply requires that we reference the "shortname" field instead of the "name" field within the template that outputs the abilities. Once that change is in place, our character sheet will begin showing the shorter names whenever they are defined.

Omitting Items from Character Sheet

As we've been doing our testing and experimenting with the character sheet, you may have noticed that there are a few items that really don't need to be included on the sheet. For example, the "Free Edge" ability for humans really doesn't need to be listed on the character sheet. Similarly, the "Power Points" edge is included in the total power points listed for the character and doesn't really need to take up space on the sheet.

We can omit selected items from inclusion on the character sheet very quickly. The Skeleton files provide a pre-defined "Print.NoPrint" tag. You can assign this tag to any thing that you don't want to have appear in the character sheet. Once assigned, you can then preclude all things possessing that tag from a given table via the List tag expression. For example, since the "oEdge" table portal currently has no List tag expression, you could define one comprising of "!Print.NoPrint" and skip all edges that possess the tag. The revised portal would like the following.

```
<portal
  id="oEdge"
  style="outNormal">
  <output_table
    component="Edge"
    showtemplate="oAbilPick">
    <list>!Print.NoPrint</list>
    <headertitle><![CDATA[
      @text = "Edges"
    </output_table>
</portal>
```

Character Sheet Phase 3 (Savage)

In this phase of developing the character sheet, we'll begin work on the right column of the sheet. This side is a bit more complex than the left side, containing both detailed sections (e.g. powers, weapons, and armor) and some interesting relationships between the different sections.

Logos

At the top of the right column of the character are the logos for both HL and the game system. These are handled automatically in the Skeleton files and there is nothing we need to do. If we wanted to stack the logos instead of placing them side-by-side, we could do that by simply configuring the orientation within the Position script for the sheet. However, we're trying to keep things compact in this character sheet, so we'll stick with the default behavior.

Eliminate Portrait Handling

The first thing we need to do for the right column is eliminate the character portrait. The Skeleton files include the output of a character portrait at the top if a portrait is specified for the character. In the interest of squeezing all the important information into the one-page sheet, we'll remove it from the sheet.

Unlike the way we handled the "Health and Power" material previously, we won't completely delete the logic for handling character portraits. We'll likely want to include the portrait in a different character layout in the future, at which point we'll want to use the logic that's provided by the Skeleton files. For this sheet, we'll simply eliminate the reference to the portrait, thereby removing it from this particular sheet without actually deleting it from the data files.

All of the logic for the portrait display resides within the "oPortrait" template. In order to remove the portrait from the character sheet, we simply need to eliminate the reference to this template from the "oRightSide" layout. This consists of deleting the "templateref" and the line of code from the Position script that places the template. Once we do that, the portrait is no longer included in the character sheet.

Edges

The next thing we need to worry about on the right side is edges. At this point, you're probably wondering why we're concerned with edges on the right side, since we already output them on the left side. The reason is that a highly complex character might not be able to fit all the edges on the left side. If there are more than will actually fit in the space at the bottom of the left column, only those that fit will be output. If that occurs, we still need to output the rest. One option for handling the extra edges is to use the second "spill-over" page for that purpose. However, edges are very important and should ideally be shown on the main character sheet, so we should output any extras at the top of the right column.

Outputting any remaining edges is actually quite easy. When generating the character sheet, HL automatically keeps track of which picks are actually output to the sheet. Each pick is output only once, which eliminates the need for you to worry about which edges have been output and which ones still need to be output. By

simply re-using the existing edges table on the right side, any edges that have not yet been output will be included.

If all of the edges are properly output on the left side, then the table of edges on the right will be empty. By using the auto-place mechanism, an empty table is completely omitted (automatically) when rendering the character sheet. So the behavior we want is handled for us automatically.

This means that all we need to do is reference the existing edges table from within the "oRightSide" layout. Doing that requires adding a "portalref" element for the table and placing the portal properly. The revised layout should look like the following.

```
<layout
  id="oRightSide">
  <portalref portal="oEdge2"/>
  <portalref portal="oGear"/>
  <position><![CDATA[
   ~position the various tables appropriately
   perform portal[oEdge2].autoplace
   perform portal[oGear].autoplace

   ~our layout height is the extent of the elements within
   height = autotop
   ]]></position>
  </layout>
```

Add a bunch of edges to the character so that it will run out of room and need to spill over to the next column, then preview the character again. Everything works as intended for edges, except we've uncovered a new problem. The table of edges on the left extends down to the bottom of the page instead of being limited to the top of the validation region. By default, the "oLeftSide" layout will extend to the bottom of the page. In order to impose the necessary limit, we need to set the bottom of the layout before we render it. This consists of adding a single line of code immediately prior to the "render" statement within the Position script of the sheet, so the block of code that configures and renders the left side layout should look like below.

```
~position the leftside layout in the upper left corner
layout[oLeftSide].width = colwidth
layout[oLeftSide].height = extent - layout[oLeftSide].top
perform layout[oLeftSide].render
```

If we wanted to be truly paranoid, we could add spill-over handling for hindrances. Theoretically, if a character has enough skills and hindrances, the list of hindrances could overrun the available space on the left side. However, the feasibility of such a character in practice is unrealistic, so we won't bother.

ARCANE POWERS

Moving downward on the right side, we have arcane powers next. Powers need to be handled a lot like weapons are handled in the provided Skeleton files. There are a number of fields that are best placed in columns within the table. As such, we'll want to use the same approach as with the weapons, wherein we use the same template as both a header and the contents of each item. This makes it possible to easily align the header elements with the contents of the table.

For the moment, though, we're just going to start with a normal table that does not possess a special header. The reason is that trying to setup everything for both the header and contents at the same time results in a more complex operation. When both are done at the same time, you have to get everything into place before you can begin testing and refining. By implementing everything in stages, we'll simplify the process for ourselves and make it possible to get the contents in good shape before we worry about the header.

There is a great deal of information that should ideally be output for arcane powers. In fact, there's so much information that it would be ridiculous to try and squeeze it all into a single line. We need to show the name, point cost, range, duration, maintenance, trappings, and description for each power. However, we're striving to keep everything on a single sheet, so we have to keep things as compact as possible. In order to achieve that goal, we're going to need our starting information to be more compact. The contents of each field that we've entered for display to the user within the UI are too large to fit in a compact space.

FIELDS FOR OUTPUT

The first thing we need to do is define a new set of fields for powers that will contain shortened versions for display on the character sheet. If these fields are non-empty, we'll use the contents, else we'll use the standard field, assuming that it's already short enough for our purposes. We'll need new fields for the point cost, range, duration, and maintenance. Open the file "miscellaneous.str" and locate the "Power" component. Copy the four existing fields for this information and give them new ids to indicate they are for use with printing. When you're done, the new fields should look similar to those below.

```
<field
  id="powPrtPts"
  name="Power Point Cost"
  type="static"
  maxlength="25">
  </field>

<field
  id="powPrtRng"
  name="Range"
  type="static"
  maxlength="25">
  </field>
<field
  id="powPrtLen"
  name="Duration"
  type="static"
  maxlength="25">
  </field>

<field
  id="powPrtMain"
  name="Maintenance"
  type="static"
  maxlength="25">
  </field>
```

Now open the file "thing_arcane.dat" and go through all of the powers. Any time you see one of the four fields with lengthy contents, define the corresponding new field with a suitably shortened version. Keep it all as brief as possible so that we can fit it all into the table for display on the character sheet.

DESIGN THE TABLE

Returning to the character sheet, we need to determine how the contents are going to look for each power. We already decided that we won't be able to squeeze everything into a single line. Now the question is whether we can limit ourselves to two lines or if we need more. So we should probably grab a piece of paper and sketch out how everything will look.

After sketching out various ideas, we'll settle on one that is compact and also includes all the most important details for each power. We'll put the name on the left in a moderate size font. We'll have a line of information that provides the point cost, range, duration, maintenance cost, and trappings. Beneath that line, we'll provide the summary information about how the power works. If a character has a dozen powers, this approach will consume a good amount of space. However, most characters with powers only have a handful of them, so this should generally work out quite well.

TABLE PORTAL

With the layout mapped out, the next step is to create a new table portal for our powers. This portal should look like many of the ones we've already defined. The key distinction is that the summary may end up being more than one line in length for some powers. As such, we need to designate the table as containing variable height items. The resulting portal should look like the following.

```
<portal
  id="oPower"
  style="outNormal">
  <output_table
    component="Power"
    showtemplate="oPowerPick"
    varyheight="yes">
    <headertitle><![CDATA[
      @text = "Arcane Powers"
    ]]></headertitle>
  </output_table>
</portal>
```

Once defined, we need to integrate the new table portal into the appropriate layout. As we've done before, this entails adding a "portalref" element for the portal and then placing the portal via the Position script. The revised "oRightSide" layout should look like below.

```
<layout
  id="oRightSide">
  <portalref portal="oEdge"/>
  <portalref portal="oPower"/>
  <portalref portal="oGear"/>
  <position><![CDATA[
    ~position the various tables appropriately
    perform portal[oEdge].autoplace
    perform portal[oPower].autoplace
    perform portal[oGear].autoplace

    ~our layout height is the extent of the elements within
    height = autotop
  ]]></position>
</layout>
```

NEW STYLES REQUIRED

We now shift our focus to the template used to display each individual power. In order to fit all the information we plan, we're going to need to use a rather small font. Looking through the various styles provided in the file "styles_output.aug", there aren't any that will serve our needs. So we'll need to create a new style that uses a small font. We'll actually need two new styles, with one centering the text and the other aligning the text to the left. This means we need to define the font separately for use by both styles. These new definition should look like the ones below.

```
<resource
  id="ofnttiny"
    face="Arial"
    size="32">
    </font>
  </resource>

<style
  id="outTiny">
  <style_output
    textcolor="000000"
    font="ofnttiny"
    alignment="center">
    </style_output>
  </style>

<style
  id="outTinyLt">
  <style_output
    textcolor="000000"
    font="ofnttiny"
    alignment="left">
    </style_output>
  </style>
```

THE TEMPLATE

We need a total of six portals in order to present all the information we selected for inclusion. The portals for showing the point cost and range will use a script that first checks the print-specific field and otherwise defaults to the normal field. The portal for duration will work the same way, with added handling for including any maintenance cost. The remaining portals will simply show the contents of an appropriate field.

We'll put the name on the left and then align the four fields across the top line within columns. The summary text will be placed beneath the line of fields, spanning the same width as the fields, and it can be more than one line in height. If the name is too large to fit, we'll allow it span two lines, since we already need at least two lines of output. Our total height will be determined after we position everything. If the name still doesn't fit, will use the built-in "sizetofit" target reference to have HL incrementally scale the font size downward until the name actually fits. We can then center the name vertically. Putting all this together yields the template shown below.

```
<template
  id="oPowerPick"
  name="Output Powers Table"
  compset="Power"
  marginvert="2">

  <portal
    id="name"
    style="outMedLt">
    <output_label
      field="name">
    </output_label>
  </portal>

  <portal
    id="points"
    style="outTiny">
    <output_label>
    <labeltext><![CDATA[
      if (field[powPrtPts].isempty = 0) then
        @text = field[powPrtPts].text
      else
        @text = field[powPoints].text
      endif
    ]]></labeltext>
  </output_label>
```

```
    </portal>

  <portal
    id="range"
    style="outTiny">
    <output_label>
      <labeltext><![CDATA[
        if (field[powPrtRng].isempty = 0) then
          @text = field[powPrtRng].text
        else
          @text = field[powRange].text
          endif
        ]]></labeltext>
      </output_label>
    </portal>

  <portal
    id="duration"
    style="outTiny">
    <output_label>
      <labeltext><![CDATA[
        var maint as string
        if (field[powPrtMain].isempty = 0) then
          maint = field[powPrtMain].text
        else
          maint = field[powMaint].text
          endif
        if (field[powPrtLen].isempty = 0) then
          @text = field[powPrtLen].text
        else
          @text = field[powLength].text
          endif
        if (empty(maint) = 0) then
          @text &= " (" & maint & ")"
          endif
        ]]></labeltext>
      </output_label>
    </portal>

  <portal
    id="trapping"
    style="outTinyLt">
    <output_label
      field="powTraps">
      </output_label>
    </portal>

  <portal
    id="summary"
    style="outPower">
    <output_label
      field="summary"
      ismultiline="yes">
      </output_label>
    </portal>

  <position><![CDATA[
    ~position all portals with the same baseline as the name;
since they use a
    ~smaller font, they will have a smaller height, so centering
them will have
    ~them appear to float up relative to the name
    perform portal[points].alignrel[btob,name,0]
    perform portal[range].alignrel[btob,name,0]
    perform portal[duration].alignrel[btob,name,0]
    perform portal[trapping].alignrel[btob,name,0]

   ~establish suitable fixed widths for the various columns of
data
    portal[name].width = 300
    portal[points].width = 100
    portal[range].width = 150
    portal[duration].width = 200

    ~position everything horizontally
    perform portal[range].alignrel[ltor,points,5]
     perform portal[duration].alignrel[ltor,range,5]
     perform portal[trapping].alignrel[ltor,duration,5]

    ~size the trappings portal to whatever space is left
     portal[trapping].width = width - portal[trapping].left

    ~size the summary portal to span the area excluding the
name
     portal[summary].left = portal[points].left
     portal[summary].width = width - portal[summary].left
     perform portal[summary].autoheight

    ~position the summary portal beneath the other line of fields
     perform portal[summary].alignrel[ttob,points,2]

    ~our height is the base of the summary portal
     height = portal[summary].bottom

    ~size the name to fit the available space, then re-center after
shrink
     perform portal[name].sizetofit[36]
     perform portal[name].autoheight
     perform portal[name].centervert
     ]]></position>

  </template>
```

Reloading the data files and previewing the character gives us a chance to review what we've got. The name and columns of data look pretty good, but the summary text beneath results in a rather jumbled look. If we output the summary text in a slightly fainter stroke and set it off against a grey background, everything would look significantly better. So we'll define a new style like the one below and use it with the summary text.

```
<style
  id="outPower">
  <style_output
    textcolor="404040"
    backcolor="e8e8e8"
    font="ofnttiny"
    alignment="left">
    </style_output>
  </style>
```

This looks very clear and readable. The amount of space available for showing the trappings of a power are small, but it's enough if the user is succinct with his descriptions of the trappings. So we're good to go for showing the powers.

Adding a Custom Header

It's now time to add the custom header to the table portal. Our header needs to label each column of information within the table. In order to easily synchronize the positions of the column headers with the actual data being output, we'll make use of a dual-purpose template. A dual-purpose template is when the same template is used for both the contents of the table and the header, and there are a number of details involved in making it all work.

The first step is to revise our table portal properly. We need to add a new "headertemplate" attribute that references the same template used as a the show template. We also need to eliminate the "headertitle" element, since we'll be defining our own header. This yields the following new portal.

```
<portal
  id="oPower"
```

```
  style="outNormal">
  <output_table
   component="Power"
   showtemplate="oPowerPick"
   headertemplate="oPowerPick"
   varyheight="yes">
   </output_table>
  </portal>
```

Next up is adding the appropriate portals to the template that will be used for the header. We need one portal for the banner that will span the full width of the table portal. We also need four additional portals for the various labels at the top of each column. For the column headers, the Skeleton files provide the "outHeader" style that is generally an excellent choice. However, we're using a much smaller font for the various fields themselves, so we should use a similar font for the headers. This results in the following new portal definitions.

```
<portal
  id="hdrtitle"
  style="outTitle"
  isheader="yes">
  <output_label
   text="Arcane Powers">
   </output_label>
  </portal>

<portal
  id="hdrpoints
  style="outTiny"
  isheader="yes">
  <output_label
   text="Cost">
   </output_label>
  </portal>

<portal
  id="hdrrange"
  style="outTiny"
  isheader="yes">
  <output_label
   text="Range">
   </output_label>
  </portal>

<portal
  id="hdrlength"
  style="outTiny"
  isheader="yes">
  <output_label
   text="Dur/Maint">
   </output_label>
  </portal>

<portal
  id="hdrtrap"
  style="outTiny"
  isheader="yes">
  <output_label
   text="Trappings">
   </output_label>
  </portal>
```

The final step is to define a Header script via the "header" element. This script is similar to the Position script, except that it's intended for positioning the various portals within the header. It is invoked after the normal Position script, which allows it to retrieve the positions of the various normal portals for the template and easily position the header portals in relation to them. Consequently, we can position each of the labels directly where they belong without any special hoop jumping. This yields the Header script below.

```
<header><![CDATA[
  ~our header height is the title plus a gap plus the header text
  height = portal[hdrtitle].height + 2 + portal[hdrpoints].height
  if (issizing <> 0) then
    done
   endif

  ~our title spans the entire width of the template
  portal[hdrtitle].width = width

  ~each of our header labels has the same width as the
corresponding data beneath
  portal[hdrpoints].width = portal[points].width
  portal[hdrrange].width = portal[range].width
  portal[hdrlength].width = portal[duration].width
  portal[hdrtrap].width = portal[trapping].width

  ~center each header label on the corresponding data beneath
  perform portal[hdrpoints].centeron[horz,points]
  perform portal[hdrrange].centeron[horz,range]
  perform portal[hdrlength].centeron[horz,duration]
  perform portal[hdrtrap].centeron[horz,trapping]

  ~align all header labels at the bottom of the header region
  perform portal[hdrpoints].alignedge[bottom,0]
  perform portal[hdrrange].alignedge[bottom,0]
  perform portal[hdrlength].alignedge[bottom,0]
  perform portal[hdrtrap].alignedge[bottom,0]
  ]]></header>
```

We can now reload the files and see how our header looks. It looks great, although we should probably include the total number of arcane power points possessed by the character within the title. That way, the total power points is conveniently grouped next to all the powers for the users. We can accomplish this by changing the "hdrtitle" portal to use a script and synthesize the appropriate information via the script. The revised portal should look similar to the one below.

```
<portal
  id="hdrtitle"
  style="outTitle"
  isheader="yes">
  <output_label>
   <labeltext><![CDATA[
    @text = "Arcane Powers (" & #trkmax[trkPower] & "
Points)"
    ]]></labeltext>
   </output_label>
  </portal>
```

CHARACTER SHEET PHASE 4 (SAVAGE)

In the previous stage of creating the character sheet, we got started on the right side and solved the most complex piece of the output in arcane powers. The remainder of the right side of the sheet introduce a few more wrinkles, but everything should proceed smoothly from this point forward.

BREAKING THE RIGHT SIDE INTO SECTIONS

The biggest remaining issue with the right side of the sheet is managing our vertical space appropriately. If a character has arcane powers, a bunch of weapons, and lots of gear, there might not be enough space on the right side in which to fit it all. It's perfectly

reasonable for that to happen, and we can safely spill the remaining items onto a second page. However, we should give thought to which information is given priority for being shown on the first page and what we're happy to let fall to the second page.

It's very important that we show the region for tracking health on the first page, since that will come into play regularly for many games. Since we're pairing the health and injuries up side-by-side, we need to ensure those get included on the first page. We should also be sure to include a reminder of any actively enabled adjustments on the first page. Weapons and armor are a secondary priority, with gear being the lowest priority for the first page.

In order to ensure that we honor these priorities, we need to carve the right side of the page up into pieces that can be placed in an appropriate sequence. So we need create a new layout that encompasses the material we'll place in the lower right corner, then we need to position it after the powers are handled. Within the remaining space, we can position a separate layout that includes weapons and armor. If we still have space left, then we can squeeze our gear into that region via a third layout.

So our next order of business is to setup these three layouts. We already have a layout for the weapons and armor that we can use. This layout is designed to split its space between the weapons and armor tables, so we don't have to do anything special to get weapons and armor to work smoothly.

For the gear, we have the table portal and simply need a layout to contain it. That layout can be defined easily as shown below, and we must remove the portal reference from the "oRightSide" layout.

```
<layout
  id="oGear">
  <portalref portal="oGear"/>
  <position><![CDATA[
    ~position the various tables appropriately
    perform portal[oGear].autoplace

    ~our layout height is the extent of the elements within
    height = autotop
    ]]></position>
  </layout>
```

The Skeleton files provide us with a table of adjustments and a layout that contains them. For simplicity, we'll just use this layout as our third layout. When we need to add injuries and such, we'll integrate them into this layout. So we don't need to do anything further with this layout.

The final thing we need to do is properly revise the sheet to incorporate these three layouts. We also need to handle these layouts within the Position script for the sheet. This entails positioning the current "oRightSide" layout, then placing the "oAdjust" layout anchored to the bottom of the sheet. Once that's done, we can position the "oArmory" layout beneath the "oRightSide" layout, using up whatever space is available. We'll place the "oGear" layout last, using the final remnants of space on the character sheet. The updated contents of the "standard1" sheet should look like the one shown below.

```
<sheet
  id="standard1"
  name="Standard character sheet, page #1">
  <layoutref layout="oLogos"/>
  <layoutref layout="oLeftSide"/>
  <layoutref layout="oAdjust"/>
  <layoutref layout="oArmory"/>
  <layoutref layout="oGear"/>
  <layoutref layout="oValidate"/>
  <position><![CDATA[
    ~set this scene variable to 1 if you want the logos to be
stacked; a value
    ~of zero places them side-by-side
    scenevalue[stacklogos] = 0

    ~setup the gap to be used between the various sections of the
character sheet
    autogap = 40
    scenevalue[sectiongap] = autogap

    ~calculate the width of the two columns of the character
sheet, leaving a
    ~suitable center gap between them
    var colwidth as number
    colwidth = (width - 50) / 2

    ~if the user wants to omit the validation report, the hide it and
allow the
    ~rest of the sheet to fill that space; otherwise, output the
layout and the
    ~top of the validation report establishes the bottom for all
other output
    var extent as number
    if (hero.tagis[source.Validation] = 0) then
      layout[oValidate].visible = 0
      extent = height
    else
      layout[oValidate].width = width
      perform layout[oValidate].render
      layout[oValidate].top = height - layout[oValidate].height
      extent = layout[oValidate].top - scenevalue[sectiongap]
      endif

    ~position the leftside layout in the upper left corner
    layout[oLeftSide].width = colwidth
    layout[oLeftSide].height = extent - layout[oLeftSide].top
    perform layout[oLeftSide].render

    ~position the logos layout in the upper right corner
    layout[oLogos].width = colwidth
    perform layout[oLogos].render
    layout[oLogos].left = width - colwidth

    ~position the activated adjustments at the bottom on the right;
this will
    ~establish the remaining space available on the right for
other layouts
    layout[oAdjust].width = colwidth
    layout[oAdjust].left = layout[oLogos].left
    perform layout[oAdjust].render
    layout[oAdjust].top = extent - layout[oAdjust].height

    ~position the rightside layout in the remaining space on the
right
    layout[oRightSide].width = colwidth
    layout[oRightSide].top = layout[oLogos].bottom +
scenevalue[sectiongap]
    layout[oRightSide].left = layout[oLogos].left
    layout[oRightSide].height = layout[oAdjust].top -
scenevalue[sectiongap] -  layout[oRightSide].top
    perform layout[oRightSide].render

    ~position the armory layout within the remaining space on the
right
    layout[oArmory].width = colwidth
    layout[oArmory].top = layout[oRightSide].bottom +
scenevalue[sectiongap]
    layout[oArmory].left = layout[oLogos].left
```

```
    ~if the top of the armory layout is below the adjustments
layout, there is no
     ~room for it, so hide it; otherwise, set height and render the
layout properly
     if (layout[oArmory].top >= layout[oAdjust].top) then
       layout[oArmory].visible = 0
     else
       layout[oArmory].height = layout[oAdjust].top -
scenevalue[sectiongap] - layout[oArmory].top
       perform layout[oArmory].render
       endif

    ~position the gear layout in whatever space is left on the right
     layout[oGear].width = colwidth
     layout[oGear].top = layout[oArmory].bottom +
scenevalue[sectiongap]
      layout[oGear].left = layout[oLogos].left

     ~if the top of the gear layout is below the adjustments layout,
there is no
      ~room for it, so hide it; otherwise, set height and render the
layout properly
      if (layout[oGear].top >= layout[oAdjust].top) then
        layout[oGear].visible = 0
      else
        layout[oGear].height = layout[oAdjust].top -
scenevalue[sectiongap] - layout[oGear].top
        perform layout[oGear].render
        endif
      ]]></position>
    </sheet>
```

Adjustments

Since the adjustments layout is always going to be given priority in the lower right corner, we should make sure it's all working now. The only time the adjustments table appears is when one or more in-play or permanent adjustments are enabled for the character when the sheet is printed. For testing, we can add a few of each and preview the character sheet. They appear nicely in the lower right corner, so there's nothing further we need to do to handle them now.

Injuries and Health

The plan is to always show any injuries in the lower right corner, along with a place to track health and power points. The injuries can be managed as a simple table portal, while the condition tracking requires that we implement something that is a little more complex. We could keep things easy by just inserting a graphic, or we could actually construct something that will work. We'll try the latter, without going overboard.

Injuries Table

Before we do that, though, we'll first get the injuries table into place. The table simply lists all injuries, and we'll need to provide a template that displays each individual injury appropriately. For the template, we just need something simple that will show the name of the injury. We can clone and adapt the template used for adjustments. This results in the table portal and template shown below.

```
 <portal
   id="oInjury"
   style="outNormal">
   <output_table
    component="Injury"
    showtemplate="oInjury">
    <headertitle><![CDATA[
      ]]></headertitle>
    </output_table>
   </portal>

 <template
   id="oInjury"
   name="Output Injuries Table"
   compset="Injury"
   marginhorz="10">

   <portal
     id="name"
     style="outNormLt">
     <output_label
      field="name">
      </output_label>
     </portal>

   <position><![CDATA[
     ~our height is the vertical extent of our portals
     height = portal[name].height
     if (issizing <> 0) then
       done
       endif

     ~size the name to fit the available space
     portal[name].width = width
     perform portal[name].sizetofit[36]
     perform portal[name].autoheight
     perform portal[name].centervert
     ]]></position>

   </template>
```

Once the table is in place, we need to add it to the layout. We'll make it 40% of the total width of the layout, leaving the rest for condition tracking. Unlike our usual approach, we cannot use "autoplace" for this table. If the table is empty, automatic placement will hide the table, which will leave an empty gap in the character sheet that will look odd. So we need to position and size the table ourselves. Since the table will use no more space than it requires, we can set the height to something very large and HL will truncate it to the appropriate height. This results in the revised layout below.

```
 <layout
   id="oAdjust">
   <portalref portal="oAdjust"/>
   <portalref portal="oInjury"/>
   <position><![CDATA[
     ~position the adjustments table at the top
     perform portal[oAdjust].autoplace

    ~position the injuries table in the left 40% beneath the
adjustments
     ~Note: We don't use autoplace, since we never want the
table hidden
     portal[oInjury].left = 0
     portal[oInjury].top = autotop + scenevalue[sectiongap]
     portal[oInjury].width = width * .4
     portal[oInjury].height = 5000

    ~our layout height is the bottom extent of the elements within
     height = portal[oInjury].bottom
     ]]></position>
   </layout>
```

If our character has no injuries, though, this looks strange. All we see is the header above the table and nothing within it. What we need is a portal that simply says "None" that we can show just below the header if the table is empty. So we'll define a new label

portal and show it appropriately via the layout. The new portal and updated layout are shown below.

```
<portal
  id="oNoInjury"
  style="outNormal">
  <output_label
    text="-None-">
    </output_label>
  </portal>

<layout    id="oAdjust">
  <portalref portal="oAdjust"/>
  <portalref portal="oInjury"/>
  <portalref portal="oNoInjury"/>
  <position><![CDATA[
    ~position the adjustments table at the top
    perform portal[oAdjust].autoplace

    ~position the injuries table in the left 40% beneath the
adjustments
    ~Note: We don't use autoplace, since we never want the
table hidden
    portal[oInjury].left = 0
    portal[oInjury].top = autotop + scenevalue[sectiongap]
    portal[oInjury].width = width * .4
    portal[oInjury].height = 5000

    ~if there are no injuries, indicate that fact
    if (portal[oInjury].itemsshown <> 0) then
      portal[oNoInjury].visible = 0
    else
      perform portal[oNoInjury].centeron[horz,oInjury]
      perform portal[oNoInjury].alignrel[ttob,oInjury,10]
      endif

    ~our layout height is the bottom extent of the elements within
    height =
maximum(portal[oInjury].bottom,portal[oNoInjury].bottom)
    ]]></position>
  </layout>
```

Health in Template

For tracking health, the standard representation in Savage Worlds is a Wounds progression from one end and a Fatigue progression from the other end. This can probably be best implemented via a handful of portals. Since the contents of those portals have no relationship to any aspects of the character, no template is truly necessary. However, it's probably easiest to manage this via a template, so that's the approach we'll use.

We need a total of eight portals within our template. We need one to indicate "Wounds" on the left and another to indicate "Fatigue" on the right. We then need the "Incapacitated" indicator as a third portal, along with three progressive increments for wounds and two for fatigue. We'll place the two labels in the upper left and upper right corners of the span within which the condition tracking is shown. The six health levels will then be arrayed across the full width of the span. This results in a template that should look like the following.

```
<template
  id="oCondition"
  name="Output Condition"
  compset="Actor">

  <portal
    id="wounds"
    style="outNameLg">
      text="Wounds">
      </output_label>
    </portal>

  <portal
    id="fatigue"
    style="outNameLg">
    <output_label
      text="Fatigue">
      </output_label>
    </portal>

  <portal
    id="wound1"
    style="outNameLg">
    <output_label
      text="-1">
      </output_label>
    </portal>

  <portal
    id="wound2"
    style="outNameLg">
    <output_label
      text="-2">
      </output_label>
    </portal>

  <portal
    id="wound3"
    style="outNameLg">
    <output_label
      text="-3">
      </output_label>
    </portal>

  <portal
    id="fatigue1"
    style="outNameLg">
    <output_label
      text="-1">
      </output_label>
    </portal>

  <portal
    id="fatigue2"
    style="outNameLg">
    <output_label
      text="-2">
      </output_label>
    </portal>

  <portal
    id="incapac"
    style="outNameLg">
    <output_label
      text="INC">
      </output_label>
    </portal>

  <position><![CDATA[
    ~position the labels at the top
    portal[wounds].top = 0
    portal[fatigue].top = 0

    ~position all health levels beneath the labels with a common
baseline
    perform portal[wound1].alignrel[ttob,wounds,10]
    perform portal[wound2].alignrel[btob,wound1,0]
    perform portal[wound3].alignrel[btob,wound1,0]
    perform portal[fatigue1].alignrel[btob,wound1,0]
    perform portal[fatigue2].alignrel[btob,wound1,0]
    perform portal[incapac].alignrel[btob,wound1,0]

    ~position the labels at the left and right edges
```

```
    portal[wounds].left = 0
    perform portal[fatigue].alignedge[right,0]

   ~put the -1 levels at each end
    portal[wound1].left = 0
    perform portal[fatigue1].alignedge[right,0]

   ~determine the remaining span over which the portals extend
    var span as number
    span = portal[fatigue1].left - portal[wound1].right

   ~tally the total width of the remaining health level portals
    var used as number
    used = portal[wound2].width + portal[wound3].width
    used += portal[fatigue2].width + portal[incapac].width

   ~divvy up the remaining horizontal space into equal pieces
and use as spacing
    ~for positioning the various health level horizontally
    var each as number
    each = span – used
    each = round(each / 5,0,0)
    perform portal[wound2].alignrel[ltor,wound1,each]
    perform portal[wound3].alignrel[ltor,wound2,each]
    perform portal[fatigue2].alignrel[rtol,fatigue1,-each]

   ~size and center the incapacitated cell in the remaining
space
    each = (portal[fatigue2].left - portal[wound3].right -
portal[incapac].width) / 2
    perform portal[incapac].alignrel[ltor,wound3,each]

   ~our height is the bottom extent of the portals
    height = portal[wound1].bottom
    ]]></position>

  </template>
```

INTEGRATION INTO LAYOUT

Now that we've got our template created, we need to integrate it into the layout. We first add a "templateref" element. Since the template doesn't actually reference any live data within the portfolio, we can hook the template up to any pick or thing we choose. In general, the best tactic in situations like this is to utilize the "actor" pick, so that's what we'll do.

Once the template is part of the layout, we then need to position it properly. It gets placed to the right of the injuries table, leaving a little bit of a gap. The top of the template should be the same as the top of the injury table. The template will calculate its own height when it is rendered. Once rendered, it's conceivable (albeit unlikely) that the template will be taller than the table, so we need to set our height to the tallest of the two visual elements. This results in the updated layout shown below.

```
 <layout
   id="oAdjust">
   <portalref portal="oAdjust"/>
   <portalref portal="oInjury"/>
   <portalref portal="oNoInjury"/>
   <templateref template="oCondition" thing="actor"
ispick="yes"/>
   <position><![CDATA[
    ~position the adjustments table at the top
    perform portal[oAdjust].autoplace

   ~position the injuries table in the left 40% beneath the
adjustments
    ~Note: We don't use autoplace, since we never want the
table hidden
    portal[oInjury].left = 0
    portal[oInjury].width = width * .4
    portal[oInjury].height = 5000

   ~if there are no injuries, indicate that fact
    if (portal[oInjury].itemsshown <> 0) then
     portal[oNoInjury].visible = 0
    else
     perform portal[oNoInjury].centeron[horz,oInjury]
     perform portal[oNoInjury].alignrel[ttob,oInjury,10]
     endif

   ~position the condition template to the right of the injuries
    template[oCondition].left = portal[oInjury].right + 50
    template[oCondition].top = portal[oInjury].top
    template[oCondition].width = width - template[oCondition].left
    perform template[oCondition].render

   ~our layout height is the bottom extent of the elements within
    height =
maximum(portal[oInjury].bottom,template[oCondition].bottom)
    ]]></position>
   </layout>
```

REFINING THE HEALTH DISPLAY

After previewing our updated character sheet, the health region looks passable, but it would really benefit by having a soft grey border around the region to set it off nicely. Unlike portals, we cannot simply put a border around a template. So we'll need to add the border via a portal within the template. This entails adding a new portal whose sole purpose is to provide the border. We then setup a margin around the edges by positioning the portals with a gap all the way around. Once that's done, we can size the border portal to encompass the full region of the other portals, resulting in a border being drawn around a collection of portals. The new portal and the revised Position script are shown below.

```
 <portal
   id="border1"
   style="outGreyBox">
   <output_label
    text=" ">
    </output_label>
   </portal>

 <position><![CDATA[
   ~leave a margin around all edges to draw our border
   var margin as number
   margin = 10

  ~position the labels at the top
   portal[wounds].top = margin
   portal[fatigue].top = margin

  ~position all health levels beneath the labels with a common
baseline
   perform portal[wound1].alignrel[ttob,wounds,10]
   perform portal[wound2].alignrel[btob,wound1,0]
   perform portal[wound3].alignrel[btob,wound1,0]
   perform portal[fatigue1].alignrel[btob,wound1,0]
   perform portal[fatigue2].alignrel[btob,wound1,0]
   perform portal[incapac].alignrel[btob,wound1,0]

  ~position the labels at the left and right edges
   portal[wounds].left = margin
   perform portal[fatigue].alignedge[right,-margin]

  ~put the -1 levels at each end
   portal[wound1].left = margin
   perform portal[fatigue1].alignedge[right,-margin]

  ~determine the remaining span over which the portals extend
```

```
    var span as number
   span = portal[fatigue1].left - portal[wound1].right

   ~tally the total width of the remaining health level portals
    var used as number
    used = portal[wound2].width + portal[wound3].width
    used += portal[fatigue2].width + portal[incapac].width

   ~divvy up the remaining horizontal space into equal pieces
and use as spacing
   ~for positioning the various health level horizontally
    var each as number
    each = span - used
    each = round(each / 5,0,0)
    perform portal[wound2].alignrel[ltor,wound1,each]
    perform portal[wound3].alignrel[ltor,wound2,each]
    perform portal[fatigue2].alignrel[rtol,fatigue1,-each]

   ~size and center the incapacitated cell in the remaining space
    each = (portal[fatigue2].left - portal[wound3].right -
portal[incapac].width) / 2
    perform portal[incapac].alignrel[ltor,wound3,each]

   ~set the first border to span the full dimensions of the health
tracker
    portal[border1].width = width
    portal[border1].height = portal[wound1].bottom + margin

   ~our height is the bottom extent of the border
    height = portal[border1].bottom
    ]]></position>
```

POWER POINTS TRACKING

The final thing we need to add to the condition tracking section is a place to mark off power points that are spent. All the various character sheets tend to show 30 boxes, so we'll do the same. We'll place a sequence of boxes beneath the health tracking region and allow the user to check boxes as points are spent. Given our limited space, we won't be able to fit 30 boxes, so we'll instead output two rows of 15 boxes each. To make tracking easier for the user, we'll ensure that every fifth box is larger and stands out from the others.

Synthesizing output like we need is most easily handled via a script. By using some nested "for/next" loops, we can construct the text for output. However, we need to figure out how we're going to output boxes without having to use bitmaps. The answer is the "Wingdings" font that is built into Windows. We can utilize the "Character Map" tool provided by Windows to view the various characters supported by each font. Only a small number of fonts are built into every installation of Windows, and these include the "Webdings" font and the "Wingdings" font. Scanning through the character map for "Wingdings", we spot a suitable box character that we can use (character code "A8" in hex).

Let's think through our logic now. We start by switching to the "Wingdings" font via the "{font Wingdings}" encoded text sequence. Between each character, we'll need to insert a tiny extra bit of spacing, which can be accomplished via the "{horz 1}" specification. For every fifth character, we need to increase the font size and then shrink it back down, which entails using "{size 52}". After every 15 characters, a line break needs to be inserted.

Since the border looks good around the health tracking, we'll use a separate border around the power points tracking region. So we'll need two new portals, with one being generating the text for output and the other providing a border. This yields the two new portals below and the additional code for the Position script given below.

```
<portal
  id="power"
  style="outPlain">
  <output_label>
   <labeltext><![CDATA[
    var i as number
    var j as number
    var k as number
    @text = "{font Wingdings}"
    for i = 1 to 2
      for j = 1 to 3
       for k = 1 to 4
         @text &= chr(168) & "{horz 1}"
         next
       @text &= "{size 52}" & chr(168) & "{size 40}"
       next @text &= "{br}"
      next
    ]]></labeltext>
  </output_label>
 </portal>

<portal
  id="border2"
  style="outGreyBox">
  <output_label
   text=" ">
   </output_label>
  </portal>

 ~position and size the power level condition
  perform portal[power].alignrel[ttob,border1,10 + margin]
  portal[power].left = margin
  portal[power].width = width - margin

 ~set the first border to span the full dimensions of the power
tracker
  portal[border2].top = portal[power].top – margin
  portal[border2].width = width
  portal[border2].height = portal[power].height + margin * 2

 ~our height is the bottom extent of the second border
  height = portal[border2].bottom
```

This doesn't provide anything impressive, but it offers a clean and effective solution in a very compact space. This will do quite nicely.

CHARACTER SHEET PHASE 5 (SAVAGE)

Overview We're making great progress and are almost finished with the character sheet. There are only a few more sections remaining, which we'll add in this phase of development.

ARMOR

The next section for us to deal with on the character sheet is armor. In Savage Worlds, armor is relatively simple, but there are still some important details we should include for each piece. Obviously, the armor rating is needed. For shields, we need to include the parry rating if it is non-zero. For armor, we need to include the areas covered by the armor.

One option is to do this with columns of data, just like we did for arcane powers. However, shields and armor need to show different details, and the number of details is small, so we'll just list the details in text appended after the name. The Skeleton files provide handling for armor that uses this same technique, so we'll adapt the provided "oArmorPick" template for our needs.

Before we do that, though, we need to consider how we're going to display the various areas covered by the armor. If we use the names "Head", "Torso", and such, we'll never be able to fit the details text

without switching to an incredibly small font or running onto a second line. Since space is at a premium, we need something to fit on one line. The easiest solution is to only show the first letter of each area, so "Head" would be shown as "H", etc. This will be immediately obvious to someone reviewing the character sheet and be very compact.

The question now becomes how best can be get the first letter of every area. Within the script, we'll be able to retrieve the names of each area, so we can then muck with the string we get back to strip out only the first characters. However, that will be a real pain to do. A much easier solution is to define an abbreviation for each of the tags for the different areas that is the first character. Once we do that, we can use the "tagabbrevs" target reference instead of "tagnames" and retrieve the abbreviations for display.

To add the abbreviations, we need to modify the various armor location tags. Open the file "tags.1st" and locate the tag group "ArmorLoc". Then add appropriate abbreviations to each tag. The result should look like the following.

```
<group
  id="ArmorLoc">
  <value id="Torso" abbrev="T"/>
  <value id="Head" abbrev="H"/>
  <value id="Arms" abbrev="A"/>
  <value id="Legs" abbrev="L"/>
  </group>
```

Now that the tags have been modified, we can shift our focus to the template showing each piece of defensive equipment. First of all, we don't need the "badstr" portal, so can rip that out. Next we can revise the "name" portal to synthesize the information we identified above for output. Since we're changing the font for the details text, sizing the "name" field to fit is inappropriate, so we must also eliminate that behavior. This yields the following revised template.

```
<template
  id="oArmorPick"
  name="Output Armor Table"
  compset="Equipment"
  marginvert="1">

  <portal
    id="equipped"
    style="outNameMed">
    <output_label>
      <labeltext><![CDATA[
        @text = "{bmpscale 3 output_armor}"
        ]]></labeltext>
      </output_label>
    </portal>

  <portal
    id="name"
    style="outNameMed">
    <output_label>
      <labeltext><![CDATA[
        ~start with the name and switch to a smaller, non-bold
font for details
        var temp as string
        @text = field[name].text
        @text &= "{size 36}{/b} ("

       ~add the defense rating
        @text &= field[defDefense].text

       ~if this is a shield, include the parry rating (if non-zero)
        if (tagis[component.Shield] <> 0) then
          @text &= ", Parry: " & signed(field[defParry].value)
          endif
        endif

       ~if this is armor, include the areas covered by the
equipment
        if (tagis[component.Armor] <> 0) then
          temp = tagabbrevs[ArmorLoc.?,","]
          if (empty(temp) = 0) then
            @text &= ", Covers: " & temp
            endif
          endif

       ~wrapup the details
        @text &= ")"
        ]]></labeltext>
      </output_label>
    </portal>

  <position><![CDATA[
    ~our height is the height of the tallest portal
    height =
maximum(portal[name].height,portal[equipped].height)
    if (issizing <> 0) then
      done
      endif

    ~if the armor is not equipped, hide the bitmap
    if (tagis[Equipped.Equipped] = 0) then
      portal[equipped].visible = 0
      endif

    ~center all portals vertically
    perform portal[equipped].centervert
    perform portal[name].centervert

    ~shift the "equipped" bitmap downward a little bit; this is
because it is a
    ~lone bitmap drawn via encoded text, and bitmaps are never
drawn within the
    ~descender portion of the text, which causes it to appear
higher than we want it
    portal[equipped].top += 4

    ~align everything horizontally
    perform portal[name].alignrel[ltor,equipped,5]
    ]]></position>

  </template>
```

WEAPONS

Weapons are next on our list. We need to decide whether we want to use columns of data for weapons or stream the info as details like we just did for armor. There are four primary pieces of info for weapons, aside from the name. These are the attack roll, damage, range (if applicable), and armor piercing rating (if any). We should also include a notation if there are special rules for the weapon that need to be remembered by the player, as well as indicating if the minimum strength requirement for the weapon is not satisfied.

We'll make use of columnar data, since that will probably look better. We'll note a failed strength requirement via a bitmap indicator the prefixes the name. We'll indicate is there are special rules with a bitmap after the name. Since we'll have limited space, we'll display the "shortname" field for each weapon. In terms of columns, we'll have one for the attack, another for damage, a third for armor piercing, and a fourth for range.

We'll use the same implementation approach as we employed for arcane powers. We first create the table without the header, and then we add the header information.

Table Portals

For the table itself, we'll first figure out what indicators to use for insufficient strength and special notes. Within the interface, we use the universal "no" symbol and a star symbol, respectively. We should strive to be consistent, so we'll use the same symbols on the character sheet. The star symbol in the interface is a bitmap that is designed for the on-screen use, so it will look poor on the character sheet. If we bring up the "Character Map" tool and scan through the various symbol fonts we discussed earlier, we can quickly find a suitable character in the "Wingdings" font.

Now that we've identified all the pieces we need, we can define the additional portals. We'll add a "special" portal to indicate special details for the weapon. We'll also add an "ap" portal to display the armor piercing rating for the weapon. For the "range" portal, we want to show something other than just blank when there is no range, so we'll revise the Label script accordingly. Lastly, we need to realize that the current portals mostly use the "outNameMed" style, which utilizes a rather large, bold font. This simply won't fit in the space we've got to work with, so we need to switch to an alternate style that will fit. We can create our own or use something that already exists. Probably the best choice is to try the "outPlain" style and see if that works, after which we can change it if there are problems. We'll use this style for all portals except for the name, which we'll leave using the "outNameMed" style.

Based on the above considerations, we'll define the portals. This yields a collection of portals that looks like the set below.

```
<portal
  id="badstr"
  style="outPlain">
  <output_label>
    <labeltext><![CDATA[
      @text = "{font Webdings}" & chr(120)
    ]]></labeltext>
  </output_label>
</portal>

<portal
  id="name"
  style="outNameMed">
  <output_label
    field="shortname">
  </output_label>
</portal>

<portal
  id="special"
  style="outPlain">
  <output_label>
    <labeltext><![CDATA[
      @text = "{font Wingdings}" & chr(171)
    ]]></labeltext>
  </output_label>
</portal>

<portal
  id="attack"
  style="outPlain">
  <output_label
    field="wpNetAtk">
  </output_label>
</portal>

<portal
  style="outPlain">
  <output_label
    field="wpDamage">
  </output_label>
</portal>

<portal
  id="ap"
  style="outPlain">
  <output_label>
    <labeltext><![CDATA[
      if (field[wpPiercing].value <> 0) then
        @text = field[wpPiercing].text
      else
        @text = "-"
        endif
    ]]></labeltext>
  </output_label>
</portal>

<portal
  id="range"
  style="outPlain">
  <output_label>
    <labeltext><![CDATA[
      if (tagis[component.WeapRange] <> 0) then
        @text = field[wpRange].text
      else
        @text = "-"
        endif
    ]]></labeltext>
  </output_label>
</portal>

<portal
  id="dots"
  style="outDots">
  <output_dots>
  </output_dots>
</portal>
```

Positioning the Portals

Now that we have all the portals, we need to position them appropriately. We'll start by centering the name. We'll also center our indicator symbols vertically, since they will be flanking the name on either side. All other portals use a smaller font than the name, so we need to align them along the same baseline as the name. Since we're using columns, we need to specify appropriate widths for each of the four columns other than the name. The name will then inherit whatever space remains. If we don't pick the correct widths for the columns, we can easily refine the values with a little bit of experimentation.

Once we have the space for the name identified, we can position the portals for each column. Now we are left with positioning the name appropriately. The two indicator bitmaps will be displayed as part of the name column. Consequently, we must determine which of the indicators is applicable and set our visibility accordingly. If the strength requirement is shown, then the name itself must be positioned to the right of the indicator.

Since the indicators are part of the space for the name, we need to subtract them from the width allocated to the name portal. Once that's done, we can automatically shrink the name to fit within the available space if it's too large. When we do this, we have to remember that shrinking the name keeps its "top" position the same, which means that smaller text will creep upwards. We want it to retain the same baseline position, so we have to re-position the portal after the resize to ensure the baseline doesn't change.

At this point, the name portal extends to the rightmost edge of the space we reserved for the name. However, we want the "special" portal to appear immediately adjacent to the name. We also want to include a sequence of dots between the name and the attack portal for better clarity. To accomplish this, we must now force the "name" portal to re-calculate its width based on the new font. Assuming the "sizetofit" operation above shrunk the name sufficiently, this will work perfectly. However, if the name is extremely long, we specified a minimum font size, so the operation may have stopped when the font size reached the minimum. If that's the case, re- calculating the size will end up making the portal wider, which will extend it into other columns. We can't allow that, so we must first get the current width, then re-calculate the width, and finally limit the new width to the original width.

Everything has been figured out now. So we can position the "name" portal properly and then position the "special" portal to its right, if necessary. The final step is showing the "dots" portal between the right edge of the used name region and the left edge of the attack column.

All of this logic can be seen in the Position script shown below.

```
 ~our height is based on the tallest portal within
  height = portal[name].height
  if (issizing <> 0) then
    done
    endif

 ~center the name and indicators vertically
  perform portal[name].centervert
  perform portal[badstr].centervert
  perform portal[special].centervert

 ~position the other portals with the same baseline as the name;
since they
  ~use a smaller font, they will have a smaller height, so
centering them will
  ~make them appear to float up relative to the name
  perform portal[attack].alignrel[btob,name,0]
  perform portal[damage].alignrel[btob,name,0]
  perform portal[ap].alignrel[btob,name,0]
  perform portal[range].alignrel[btob,name,0]
  perform portal[dots].alignrel[btob,name,0]

 ~establish suitable fixed widths for the various columns of data
  portal[attack].width = 135
  portal[damage].width = 165
  portal[ap].width = 50
  portal[range].width = 225

 ~assign the name the remaining horizontal space
  ~Note: We must also account for a gap of 5 between portals.
  portal[name].width = width - 4 * 5
  portal[name].width -= portal[attack].width +
portal[damage].width
  portal[name].width -= portal[ap].width + portal[range].width

 ~position our columns now, except for the name
  portal[attack].left = portal[name].right + 5
  perform portal[damage].alignrel[ltor,attack,5]
  perform portal[ap].alignrel[ltor,damage,5]
  perform portal[range].alignrel[ltor,ap,5]

 ~setup to track our position when positioning portals along the
x-axis
  var x as number
  x = 0

 ~if the weapon satisfies the minimum strength requirement, hide
the bitmap,
  if (tagis[Helper.BadStrReq] = 0) then
    portal[badstr].visible = 0
  else
    portal[badstr].left = 0
    x = portal[badstr].right
    endif

 ~if there are no special details for the weapon, hide the bitmap
  if (field[wpNotes].isempty = 1) then
    portal[special].visible = 0
    endif

 ~shrink the space for the name based on the presence of the
two bitmaps
  portal[name].width -= portal[badstr].width * portal[badstr].visible
  portal[name].width -= portal[special].width *
portal[special].visible

 ~size the name to fit the available space, then reposition it at
the baseline
  ~Note: This is needed since smaller text will have the same top
position.
  perform portal[name].sizetofit[36]
  perform portal[name].autoheight
  perform portal[name].alignrel[btob,attack,0]

 ~recalculate the width of the name based on the sized font
  ~Note: This is needed so we can determine the span for the
row of dots.
  ~Note: We must also cap the name portal to its previous width,
just in case
  ~       the "sizetofit" didn't shrink the name far enough to fully fit.
  var limit as number
  limit = portal[name].width
  perform portal[name].autowidth
  if (portal[name].width > limit) then
    portal[name].width = limit
    endif

 ~position the name and special details indicator portals
horizontally now
  portal[name].left = x  x = portal[name].right
  if (portal[special].visible <> 0) then
    portal[special].left = x
    x = portal[special].right
    endif

 ~extend the dots from the right of the name across to the attack
portal
  if (x > portal[attack].left - 10) then
    portal[dots].visible = 0
  else
    portal[dots].left = x + 5
    portal[dots].width = portal[attack].left - 5 - portal[dots].left
    endif
```

Adding the Header

Our table is looking good, so we can now deal get the header working correctly. We already have three header portals that we'll want to retain. These portals are being positioned relative to the portals in the main table, so they are working perfectly. All we need to do is add another header portal for the armor piercing rating. This can be added and then positioned just like the other header portals in the Header script.

Looking at it all, though, the header labels appear to be a little bit larger than would be optimal. Since we're using a smaller font than the provided Skeleton files, we need to ensure that the header is shrunk a bit, too. We can accomplish this by either switching to a new style or simply changing the font size used for font in the existing style. We'll do the latter and shrink the font a full point (from 36 to 32). Once we shrink the font size, we can also change

the column headers to show the proper name instead of an abbreviation.

Putting it all together, this yields the following portals and Header script for the template.

```
<portal
  id="hdrtitle"
  style="outTitle"
  isheader="yes">
  <output_label
    text="Weapons">
    </output_label>
  </portal>

<portal
  id="hdrattack"
  style="outHeader"
  isheader="yes">
  <output_label
    text="Attack">
    </output_label>
  </portal>

<portal
  id="hdrdamage"
  style="outHeader"
  isheader="yes">
  <output_label
    text="Damage">
    </output_label>
  </portal>

<portal
  id="hdrap"
  style="outHeader"
  isheader="yes">
  <output_label
    text="AP">
    </output_label>
  </portal>

<portal
  id="hdrrange"
  style="outHeader"
  isheader="yes">
  <output_label
    text="Range">
    </output_label>
  </portal>

<header><![CDATA[
  ~our header height is the title plus a gap plus the header text
  height = portal[hdrtitle].height + 2 + portal[hdrattack].height
  if (issizing <> 0) then
    done
    endif

  ~our title spans the entire width of the template
  portal[hdrtitle].width = width

  ~each of our header labels has the same width as the
corresponding data beneath
  portal[hdrattack].width = portal[attack].width
  portal[hdrdamage].width = portal[damage].width
  portal[hdrap].width = portal[ap].width
  portal[hdrrange].width = portal[range].width

  ~center each header label on the corresponding data beneath
  perform portal[hdrattack].centeron[horz,attack]
  perform portal[hdrdamage].centeron[horz,damage]
  perform portal[hdrap].centeron[horz,ap]
  perform portal[hdrrange].centeron[horz,range]

  ~align all header labels at the bottom of the header region
  perform portal[hdrdamage].alignedge[bottom,0]
  perform portal[hdrap].alignedge[bottom,0]
  perform portal[hdrrange].alignedge[bottom,0]
  ]]></header>
```

GEAR

The final piece of the character sheet is handling the gear. The Skeleton files provide us with a gear table that lists all the gear in a single column. Since a single column is used, the font is large and bold, plus there is plenty of unused space. We need something compact, using a smaller, non-bold font, and employing at least two columns. In fact, if we can switch to a significantly smaller font, we can probably even squeeze three columns in, which would be ideal. So we'll start by modifying the table portal to use three columns.

Looking at the existing template, we don't want to keep the quantity in a separate column from the actual gear name. So we'll eliminate the separate "value" portal, but we'll move the logic for generating the quantity into the "name" portal. The template uses a large horizontal margin, which we need to shrink to something that will be just enough to separate our three columns. A value of 5 ought to be about right.

We need to use a small font to squeeze the gear into three columns, so we'll simply define a style specifically for use with gear. In the file "styles_output.aug", we'll create the "outGear" style and a corresponding font for use with the style. We'll use a small point size for the font and not use bold. This results in the new definition shown below.

```
<style
  id="outGear">
  <style_output
    textcolor="000000"
    font="ofntgear"
    alignment="left">
    </style_output>
  <resource
    id="ofntgear">
    <font
      face="Arial"
      size="36">
      </font>
    </resource>
  </style>
```

We can now implement the portal appropriately. We'll use the new style and prepend the quantity onto the name, which yields the following portal.

```
<portal
  id="name"
  style="outGear">
  <output_label>
    <labeltext><![CDATA[
      if (stackable = 0) then
        @text = ""
      elseif (field[stackQty].value = 1) then
        @text = ""
      else
        @text = field[stackQty].text & "x "
        endif
      @text &= field[name].text
      ]]></labeltext>
    </output_label>
  </portal>
```

With the portal in place, we need to position it. The name portal is sized based on the text within it, so our first action is the limit the width of the portal to the width of the template. Once we do that, we can try shrinking the font size to better fit longer names into the available space. Some names may still be too long, so we now limit the height to a single line. Lastly, we align the portal at the bottom of the template, which ensures that all gear entries across a row share the same baseline. This yields the following Position script.

```
~our height is the height of the tallest portal
 height = portal[name].height
 if (issizing <> 0) then
   done
   endif

~limit the width of the name to the width of the template
 if (portal[name].width > width) then
   portal[name].width = width
   endif

~size the name to fit the available space, if needed
 perform portal[name].sizetofit[30]

~limit the name to a single line based on the updated font size
 portal[name].lineheight = 1

~align the name at the bottom of the template
 perform portal[name].alignedge[bottom,0]
```

This looks pretty good, except that there is a variety of gear that has names longer than will fit in the space we've got. The solution is to do for all gear what we've already done for weapons. We can designate the "Gear" component as supporting short names, allowing a shorter name to be defined for every piece of equipment. Then we can change the Label script for the "name" portal to reference the "shortname" field instead of the "name" field. Now we should be in great shape for handling gear in a very compact fashion.

Character Sheet Refinement (Savage)

Our basic character sheet is complete. Now we need to take a look at it and determine what is missing and/or needs to be refined.

Gap on Right

One mildly annoying facet of our character sheet is the gap that appears on the right side beneath the gear table. We need to position the adjustments and condition layout at the bottom in order to determine how much space we have left for everything else. However, if everything else doesn't need all that space, we're left with a gap that looks rather odd. It would probably look better if the layout at the bottom we positioned directly beneath the gear.

Fortunately, we can accomplish this very easily. After the gear layout is rendered within the Position script of the sheet, we can check to see if there is any open space beneath it. If there is open space, we can move the adjustments layout upwards so that it is placed immediately beneath the gear layout. This is achieved by adding the following lines of code to the very end of the Position script.

```
 ~if there is open space beneath the gear layout, move the
adjustments upward
  if (layout[oAdjust].top > layout[oGear].bottom +
scenevalue[sectiongap]) then
    layout[oAdjust].top = layout[oGear].bottom +
scenevalue[sectiongap]
    endif
```

Wait a minute. That's not going to work all the time. What if the gear layout doesn't fit? Looking back at all the special handling we've got for possibly omitting the gear and maybe the armory layouts, we have to handle all those cases for closing the gap. This entails extending our logic to the three different cases: gear layout present, gear layout omitted but armory layout present, and neither layout present. The revised script code should look like the following.

```
 ~if the gear layout is visible and there is open space beneath it,
move the
  ~adjustments layout upward
  if (layout[oGear].visible <> 0) then
    if (layout[oAdjust].top > layout[oGear].bottom +
scenevalue[sectiongap]) then
      layout[oAdjust].top = layout[oGear].bottom +
scenevalue[sectiongap]
      endif

 ~otherwise, do the same check with respect to the armory
layout
  elseif (layout[oArmory].visible <> 0) then
    if (layout[oAdjust].top > layout[oArmory].bottom +
scenevalue[sectiongap]) then
      layout[oAdjust].top = layout[oArmory].bottom +
scenevalue[sectiongap]
      endif

 ~otherwise, position relative to the right-side layout  else
    if (layout[oAdjust].top > layout[oRightSide].bottom +
scenevalue[sectiongap]) then
      layout[oAdjust].top = layout[oRightSide].bottom +
scenevalue[sectiongap]
      endif
    endif
```

Weapon Attacks

One thing we overlooked was the way we present the attack values for weapons. For attributes and skills, we actually show the proper die-type bitmap. However, weapon attacks simply show a text version. It would be perfectly reasonable to change the weapon attacks to show the proper die-type bitmaps instead, but it's also reasonable to use the text version. There's no "right" answer here, and switch to the die-type bitmaps will result in each weapon entry requiring a bit more space, both horizontally and vertically. Since we already have a limited amount of space for the weapon names and our vertical space is somewhat constrained as well, we're probably better off sticking with the text version. So that's what we'll go with for our finished character sheet.

Spillover Sheet

The question remains regarding what happens when material doesn't fit on the first page of the character sheet. The Skeleton files provide us with a spillover sheet for this purpose, but we haven't done anything with it yet. Now is an excellent time to review it, and we'll find it in the file "sheet_standard2.dat".

Looking at the existing sheet, it already includes armor, weapons, and gear. It simply re-uses that same table portals that are used on the first sheet, so any excess items that don't fit on the first page will be output in the same way on the second page.

We need to assess if there is anything else that might be dropped from the first page and need to spill onto the second page. While it is highly unlikely, there is one thing that could be spilled over:

arcane powers. If a character has a huge number of powers, or if a character has lots of edges that wrapped to the right side and then there isn't room for all the powers, we could end up with powers needing to spill over. So we need to properly handle that within the second sheet.

This is easy to do. All that's required is to add a "portalref" element to the "oStandard2" layout that is used for the second page. Once that's done, we simply insert an appropriate automatic placement statement in the Position script for the new portal. The revised layout should look like the one below.

```
<layout
  id="oStandard2">
  <portalref portal="oPower"/>
  <portalref portal="oArmor"/>
  <portalref portal="oWeapon"/>
  <portalref portal="oGear"/>
  <position><![CDATA[
    ~position the various tables in the desired sequence
    perform portal[oPower].autoplace
    perform portal[oArmor].autoplace
    perform portal[oWeapon].autoplace
    perform portal[oGear].autoplace

   ~the height of the layout is the bottommost extent of the
elements within
    height = autotop
    ]]></position>
  </layout>
```

That's all there is to it. We've now got full handling in place for anything that might not fit on the first page of the character sheet.

STATBLOCK OUTPUT (SAVAGE)

Most game systems have a standard format for presenting a character in a relatively compact, text-only format. The more commonly used term for this format is a "statblock". You'll find NPCs and monsters generally presented using this format within the various rulebooks. Savage Worlds has such a format, so we'll now make sure that we generate the appropriate output.

THE GAME PLAN

The statblock format we'll use for Savage Worlds can be found in the Bestiary section of the core rulebook. It's a simple text format that makes use of bold text for important names. While most creatures in the Bestiary don't possess any gear, there are a few that do. We'll be using the way those entries are formatted as a guideline for how we should be synthesizing our statblock output.

All statblock output is generated via a Synthesize script within a "dossier" element. We're going to start with the statblock that is provided by the Skeleton data files. This starting point can be readily adapted for our purposes with Savage Worlds. It can be found within the file "out_statblock.dat", which also includes a number of procedures that are called directly from the Synthesize script.

HOW OUTPUT WORKS

As mentioned above, text output is generated via a Synthesize script. Since text output is really just a single big stream of text, it's very clunky to require the author to keep remembering to append data to a string variable. It's also very inefficient from a scripting standpoint. Consequently, the Kit provides a special mechanism that is is specific to text output. The text being synthesized is managed internally by HL and the "append" statement allows the author to easily tack more material onto the end of the output. This results in a simplified process for authors, although it also requires the output be generated in a serialized fashion.

The general mechanism is designed to support various different types of output, including plain text, HTML, and BBCode. In order to make this easy for authors, there are a number of different special symbols that are pre-defined by the Kit. Based on the output format chosen by the user, these symbols map to the appropriate codes for that format. For example, the "@boldon" symbol is used to enable bold text within the output. When the user selects HTML output, this symbol maps to "{b}", while it is empty for plain text output. This allows authors to synthesize the output once and let HL do the work of tailoring it to the appropriate format.

The following sequence of script code will output a series of attributes as text, where each is listed on a new line, the name of the field is in bold, and the description is not.

```
foreach pick in hero where "component.Attribute"
  append @boldon & eachpick.field[name].text & @boldoff
  append ": " & eachpick.field[description]
  append @newline
  nexteach
```

OUTPUTTING THE BASICS

The mechanism provided by the Skeleton files generates all of the basic details for a generic statblock. We're going to adapt it to the specifics of Savage Worlds, and we'll start with the attributes, skills, and derived traits.

The sample script includes the age as an example of inserting personal information about the character. The Savage Worlds format does not include such data, so we need to omit it. That means deleting the code that outputs the age.

For attributes and skills, a procedure is used that is passed in the tag expression to select the proper picks. In both uses, we need to modify the tag expression to omit the attributes and skills that are hidden from output. This results in the revised code below.

```
~output attributes
 append @boldon & "Attributes: " & @boldoff
 tagexpr = "component.Attribute &  !Hide.Attribute"
 call sbtraits

~output skills
 append @boldon & "Skills: " & @boldoff
 tagexpr = "component.Skill &  !Hide.Skill"
 call sbtraits
```

Now we need to determine whether to revise the procedure contents. The current procedure appears to work perfectly for attributes and skills, so there is are no changes required.

Attributes and skills are now handled, so that leaves derived traits. Savage Worlds uses a different format for derived traits, so we need to implement them separately. Since the format is only used for derived traits, we can either write a procedure that does it or put the code directly into the Synthesize script. If the Synthesize script was large and complex, it would probably we worth using a separate procedure, but the script is relatively simple, so we'll just insert the code directly.

The Skeleton files provide logic for outputting derived traits after special abilities. We'll start by moving that code up to just below the output of skills. Once that's done, we'll adapt the code. Since Savage Worlds uses a comma-separated list for derived traits, we need to track when we need to insert a comma. We'll use the same technique as the various procedures, which have an "ismore" variable that starts at zero and gets changed to one after something is output. If the variable is non-zero, we need to insert a comma before the next item. This yields the following block of code.

```
~output derived traits
 var ismore as number
 ismore = 0  foreach pick in hero where "component.Derived &
!Hide.Trait" sortas explicit
   if (ismore <> 0) then
     append ", "
     endif
   append @boldon & eachpick.field[name].text & ": " &
@boldoff
   append eachpick.field[trtDisplay].text
   ismore = 1
   nexteach
 append @newline
```

GEAR OUTPUT

After all of the traits are output, the Savage Worlds format includes whatever gear the character possesses. All weapons, armor, and miscellaneous gear are lumped into this one block of material. In the case of weapons, the damage and range are shown in parentheses. In the case of armor, we'll include the defense rating and any parry bonus in parentheses. Simple equipment will be listed by name, along with any quantity possessed.

The first thing we need to do is handle how all the gear will be assembled for output. We need a single, comma-separate list, but we also need to synthesize each type of gear differently using separate procedures. To deal with this, we need to build up our gear list as a string. Once the final string is constructed, we can then output it. This results in the logic below.

```
~synthesize all gear
 var details as string
 details = ""
 call sbweapons
 call sbarmor
 call sbgear
 if (empty(details) = 0) then
   append @boldon & "Gear: " & @boldoff & details & @newline
   endif
```

Now we need to go into each of the procedures we rely upon and make them work the way they should. Within each of the procedures, we first need to setup the "details" variable for use as a parameter. The Synthesize script defines "details" and initializes it to empty. Each procedure will then append its items to the end of the "details" variable. This way, each procedure will build on the results of the preceding procedures.

Each procedure must also determine whether to start with a comma before the first item. Each procedure maintains it's own notion of whether a comma is needed, and its state must be initialized based on whether we already have anything within the "details" variable. If "details" is non-empty, we assume we'll need a comma before the first item we add in the procedure.

We can now adapt the existing procedures for weapons and armor to generate the text as we outline above. We can also add a new procedure for basic equipment, which will use the "grStkName" field to incorporate the quantity information. This results in the three procedures below.

```
 <procedure id="sbweapons"
scripttype="synthesize"><![CDATA[
    var details as string
    var ismore as number
    ismore =  !empty(details)
    ~output a list of all weapons
    foreach pick in hero where "component.WeaponBase" sortas
Armory
      if (ismore <> 0) then
        details &= ", "
        endif
      details &= eachpick.field[name].text
      details &= " " & eachpick.field[wpNetAtk].text
      details &= " (" & eachpick.field[wpShowDmg].text
      if (eachpick.tagis[component.WeapRange] <> 0) then
        details &= ", " & eachpick.field[wpRange].text
        endif
      details &= ")"
      ismore = 1
      nexteach
    ]]></procedure>

 <procedure id="sbarmor" scripttype="synthesize"><![CDATA[
    var details as string
    var ismore as number
    ismore =  !empty(details)
    ~output the details of all armor
    foreach pick in hero where "component.Defense" sortas
Armory
      if (ismore <> 0) then
        details &= ", "
        endif
      details &= eachpick.field[name].text
      details &= " (" & signed(eachpick.field[defDefense].text)
      if (eachpick.tagis[component.Shield] <> 0) then
        if (eachpick.field[defParry].value <> 0) then
          details &= ", Parry" & signed(eachpick.field[defParry].text)
          endif
        endif
      details &= ")"
      ismore = 1
      nexteach
    ]]></procedure>

 <procedure id="sbgear" scripttype="synthesize"><![CDATA[
    var details as string
    var ismore as number
    ismore =  !empty(details)
    ~output the list of all gear
    foreach pick in hero where "component.Equipment"
      if (ismore <> 0) then
        details &= ", "
        endif
      details &= eachpick.field[grStkName].text
      nexteach
    ]]></procedure>
```

SPECIAL ABILITIES

The next component of statblock output is the various special abilities for the character. All of the abilities are lumped together within the statblock, including edges, hindrances, and racial abilities. We could use a similar approach to gear for abilities, but all abilities are simply output with a name and summary, so can use a single mechanism for all abilities.

We only want to output special abilities if we have at least one. Fortunately, we can easily detect this. Since all abilities derived from the shared "Ability" component, we can check to see if the hero contains any picks that possess the "component.Ability" tag. If so, then we know that we have at least one ability to output and can do so. We'll output the various abilities via a called procedure, although we could just as easily include the code directly here. This results in the following code for orchestrating the output of abilities.

```
~output special abilities
 if (hero.haschild["component.Ability"] <> 0) then
   append @boldon & "Special Abilities:" & @boldoff &
@newline
   call sbability
   endif
```

Lastly, we need to implement the procedure that outputs the actual abilities. Since we have all of the abilities lumped into one list, we should organize it appropriately to show racial abilities, edges, and hindrances together. The most obvious way to do this is to use three different "foreach" loops, but there is an easier way. If we use a single "foreach" loop on all abilities and sort the sequence appropriately, we can get the desired order. We already have a sort set that will work perfectly for us, and that's the "SpecialTab" sort set. This sort set organizes everything by type, and since we only have abilities in our list, those abilities will be sorted just the way we want them.

When outputting the individual special abilities, the Savage Worlds format indents each ability. Unfortunately, there is no reliable way to indent with the various different text formats we need to support. So the easiest solution is to simply ignore the indentation and otherwise output the abilities using the same presentation style.

Putting this all together yields a procedure that looks the one below.

```
<procedure id="sbability" scripttype="synthesize"><![CDATA[
  ~output a list of all abilities
  foreach pick in hero where "component.Ability" sortas
SpecialTab
    append chr(149) & eachpick.field[shortname].text & ": "
    append eachpick.field[summary].text & @newline
    nexteach
  ]]></procedure>
```

That's incredibly simple. Because of it's simplicity, we're probably better off eliminating the extra procedure and just putting the logic directly into the Synthesize script. So we'll move the logic into the script, after which we can delete the procedure. The revised code within the Synthesize script should now look like below.

```
~output special abilities
 if (hero.haschild["component.Ability"] <> 0) then
   append @boldon & "Special Abilities:" & @boldoff &
@newline
   foreach pick in hero where "component.Ability" sortas
SpecialTab
    append chr(149) & eachpick.field[shortname].text & ": "
    append eachpick.field[summary].text & @newline
    nexteach
   endif
```

ARCANE POWERS

The final element of statblock output that we're missing is arcane powers. We can handle the arcane powers in the exact same way as special abilities. The key differences are the tag expression, we don't have to worry about sorting different types of powers, and there is no "shortname" field for powers. We make those changes and end up with the following code for outputting arcane powers.

```
~output arcane powers
 if (hero.haschild["component.Power"] <> 0) then
   append @boldon & "Arcane Powers:" & @boldoff & @newline
   foreach pick in hero where "component.Power"
    append chr(149) & eachpick.field[name].text & ": "
    append eachpick.field[summary].text & @newline
    nexteach
   endif
```

ASSORTED REMAINING ELEMENTS

MISCELLANEOUS CLEANUP (SAVAGE)

Our data files are finally nearing completion. This would be an excellent opportunity for us to stop and perform an assessment of what's been completed within our data files and what we still need to do. While doing this, it's also prudent to go through how everything works and identify any facets that we want to refine further. The goal is to assemble an updated "to do" list that will guide us through the final stages of getting our data files ready for release.

After a bit of testing and experimentation, we can put together a pretty good list. Based on our status assessment, we've got a lot of little things to tweak, so we might as well get most of them handled and removed from our list.

SHOWING ATTRIBUTE POINTS

Within the title above most of our tables, we show the number of points that have been used and any that are left or overspent. However, there is one place where we don't do this and should. The table of attributes on the "Basics" tab don't provide this helpful feedback to the user. Adding it is a simple matter of changing the contents of the HeaderTitle script to work like the other tables. So we modify the HeaderTitle script within the "baAttrib" table portal to look like the following.

```
<headertitle><![CDATA[
  @text = "Attributes  -  " &
hero.child[resAttrib].field[resSummary].text
  ]]></headertitle>
```

POWER POINTS TRACKER

The "Power Points" tracker is always appearing on the "In-Play" tab, regardless of whether the character possesses an arcane background. We need to ensure that it only appears when a suitable arcane background is possessed. That would be any arcane background except for "Weird Science", since gizmos each have their own power points to be tracked.

There are three steps in applying this fix. The first is to modify the "ipTracker" portal that shows trackers to use a List tag expression. The Skeleton files pre-define a "Hide.Tracker" tag that we can use, so we change the tag expression to exclude any trackers that are expressly hidden.

The second step is to assign the "Hide.Tracker" tag to the "trkPower" thing. This will ensure that the tracker never appears in

the table unless we specifically make it visible. We can make it visible by deleting the tag.

The final step is to modify the "Arcane" component to delete the tag. Since we want to delete the tag for all arcane backgrounds, we'll delete the tag within an Eval script on the component. That way, all arcane backgrounds inherit the behavior. We actually want to suppress the tracker for the "Weird Science" background, which we can identify via the "Arcane.WeirdSci" tag, so we'll have a special exception. This results in the script below.

```
<eval index="3" phase="Render" priority="5000"><![CDATA[
  if (tagis[Arcane.WeirdSci] = 0) then
    perform hero.child[trkPower].delete[Hide.Tracker]
    endif
  ]]></eval>
```

The tracker should now be hidden unless an appropriate arcane background is selected.

Abilities on "Special" Tab

The "Special" tab contains a list of all facets of the character that are designated for inclusion on the tab. After the name of an entry, the nature of that entry is shown within parentheses. For the various special abilities, they show two different natures. For example, edges will show both "Edge" and "Ability", while hindrances will show both "Hindrance" and "Ability". They should only be showing a single nature, and the "Ability" designation is extraneous.

If we take a look at the way the "Special" tab contents are handled, there is a "source" portal within the template. This portal uses the "tagnames" target reference and requests all tags belonging to the "SpecialTab" tag group. So the problem is that all of our abilities are being assigned two different "SpecialTab" tags.

When we modified the "Ability" component to be shared among the three different component sets (edges, hindrances, and racial abilities), we overlooked this behavior. Since the "Ability" component is shared, it should not possess its own "SpecialTab" tag. So the solution is to delete the "SpecialTab.Ability" tag from the file "tags.1st" and then modify the "Ability" component to no longer possess the tag. Once we make these two changes, abilities on the "Special" tab appear with only a single nature associated.

Names of Tracked Resources

We can now return to the "In-Play" tab to look at the presentation of tracked resources within the "ipTracker" template. Each resource is shown with its full name in a bold font. Many of the items shown in the resource list have names that are longer than will fit, so we need to handle them appropriately to maximize the chance that they do fit.

There are two different techniques that we can use, and there's no reason why we can't use both. The first thing is to realize that the majority of the items shown within the resource table possess a shorter name via the "shortname" field. Not all of them have this field, but we'll use it if it exists. This can be integrated by change the "name" portal to use a Label script, wherein the script checks for the presence of the "shortname" component and uses the field if available. The revised portal is shown below.

```
<portal
  id="name"
  style="lblNormal">
  <label>
    if (tagis[component.shortname] <> 0) then
      @text = field[shortname].text
    else
      @text = field[name].text
      endif
    ]]></labeltext>
  </label>
</portal>
```

The second change is that we can shrink the font used for the name if the name doesn't fit. This entails using the "sizetofit" target reference on the portal, after which we must re-center the portal. The script code shown below can be added at the end of the Position script to accomplish this.

```
~shrink the name portal if it doesn't fit, then re-center after the
shrink
  perform portal[name].sizetofit[32]
  perform portal[name].centervert
```

At this point, we'll be doing the best job we can of squeezing resources into the table for display.

Simplify Encumbrance Display

Another item on our task list is to modify the display of encumbrance. If the character has items that have a fractional weight, the total encumbrance shown has up to two decimal places, which just looks wrong. What we need is to show the encumbrance as a simple integer value.

If we show an integer value, the first instinct would be to round the value off normally. That would mean that a value of 14.4 would be rounded off to 14 and a value of 14.5 would become 15. While this works great in concept, it fails in a very important way. If the character maximum load is 40 and the character has gear totaling 40.1 pounds of weight, the encumbrance exceeds the limit and will be flagged as a problem, but the total encumbrance will be shown as 40 with normal rounding. To avoid problems like this, we need to always round the value upwards, so a value of 40.1 becomes 41 when rounded.

We only need to worry about the rounding for the value that is actually displayed to the user. The internal value should be maintained with maximum accuracy. The value displayed is synthesized within the second Eval script for the thing "resEncumb". We can revise this script to round the value before displaying it, which looks like the following.

```
~show the total weight carried and the maximum for the current
load level
  var spent as number
  spent = round(field[resSpent].value,0,1)  field[resShort].text =
spent & " / " & field[resMax].value
```

Encumbrance of Stacked Items

The mechanism for tallying encumbrance appears to have a problem. Items that possess a quantity of one are being added correctly to the overall encumbrance total. However, an item with a quantity greater than one is only adding its weight a single time. If the quantity is greater than one, the total accrued weight must be the individual weight of the item times the total quantity possessed.

The accrual of weight for encumbrance is handled within the second Eval script for the "Gear" component. Fixing this problem entails properly recognizing when the quantity is more than one and multiplying by the quantity. This can be resolved by modifying the Eval script to the one shown below.

```
 ~if this piece of gear is held by something else, ignore it - we'll
get it below
  if (isgearheld <> 0) then
    done
    endif

 ~if this piece of gear is a topmost holder, ignore it - it's not on
the character
  if (tagis[thing.holder_top] <> 0) then
    done
    endif

 ~determine the net weight of the gear, being sure to multiply by
the quantity
  var weight as number
  weight = field[gearNet].value
  if (stackable <> 0) then
    if (field[stackQty].value <> 1) then
      weight *= field[stackQty].value
     endif
    endif

 ~accrue the net weight of this piece of gear into the
encumbrance resource
  perform #resspent[resEncumb,+,weight,field[name].text]
```

Verify Trappings Are Changed

Each arcane power begins with trappings that list assorted possibilities from the rulebook. The user is expected to edit the trappings and specify the actual behaviors for each power. At present, there is no check in place for verifying it has been changed.

While we can't verify that the trappings are valid (it's arbitrary text), we can verify that the user has at least modified the field from its original contents. This is achieved via use of the "ischanged" target reference on the field.

The best way to handle this is by defining a new Eval Rule. If the rule is not satisfied, then a suitable error message can be output to the validation report. To ensure that the rule is applied equally to all arcane powers, we'll add the rule to the "Power" component. This results in the new Eval Rule below being defined.

```
 <evalrule index="1" phase="Validate" priority="5000"
      message="Trappings must be specified for the
power"><![CDATA[
   if (field[powTraps].ischanged <> 0) then
     @valid = 1
     endif
   ]]></evalrule>
```

Names Within Summary Panels

There are a number of places within the summary panels where the names shown extend past the edge of the panel. It would be much better if the names were to fit fully within the available space. One option would be to widen the summary panels, but that would reduce the number of visible panels to one, so that's not a good idea. Many of the items shown within the summary panels have been converted to support "short names". As such, a better solution would be to make use of the shorter name within the summary panels whenever one is available.

Switching to the "shortname" field is quick and easy within the summary panels. There are four templates that can be switched. These templates show abilities, traits, weapons, and defensive equipment. Since all of these different items support "short names", we can safely change each of the templates to reference the "shortname" field instead of the "name" field. After that, the shorter name will be shown whenever one exists.

Arcane Power Names

Within Savage Worlds, players are encouraged to come up with their own names for the arcane powers possessed by their characters. The current data files don't allow for this. So we need to allow users to name the arcane powers for a character. Given that an arcane power could have a name that is too long to fit in the available space on the "Arcane" tab, we need shrink the name if necessary. In addition, it would ideal to show the original name of the power in parentheses beneath the user name and with a softer presentation so that it is clearly secondary to the user-assigned name.

Designating arcane powers as namable by the user is the first step. This is accomplished by assigning the "usernamable" attribute for the "Power" component. As soon as this attribute is set to "yes", all powers can thereafter be named by the user via the right-click menu.

The next step entails adding the original name in parentheses beneath the user-assigned name. The "name" portal within the "apPower" template displays the name for an arcane power. One option is to change the portal to be multi-line and use a Label script to synthesize the output. This script can start with the current name and append the original name only if the user has named the power. The problem with that approach is that we can't use the "sizetofit" mechanism on such a portal. So we're better off creating a second portal and managing the two.

In order to add the second portal, we'll need to define a new style. We need to use a font that is left-aligned, of moderate size, not bold, and in a less prominent color. We can define just such a style as shown below.

```
<style
  id="lblOldName">
  <style_label
    textcolor="808080"
    font="fntsummary"
    alignment="left">
    </style_label>
  </style>
```

Now that the style is in place, we can define the new portal. It needs to utilize a Label script so that the original name can be placed within parentheses. Other than that, it's extremely simple and looks like the portal shown below.

```
<portal
  id="original"
  style="lblOldName">
  <label>
    <labeltext><![CDATA[
      @text = "(" & field[thingname].text & ")"
      ]]></labeltext>
    </label>
  </portal>
```

The final task is to revised the Position script to properly position the two portals. With the script, we also need to utilize the "sizetofit" target reference on both portals. This will ensure that long names, whether the original power name or the user-assigned name, will fit appropriately into the space available. Since the majority of power names are shorter than the space we've currently allocated, we'll also shrink that space a little bit (from 200 to 175). The updated script is presented below.

```
 ~set up our height based on our tallest portal
  ~Note: Since the portal is a script-based label, it will not have
any contents
  ~when the height of the template is initially determined for
general sizing.
  ~Consequently, we can't use "textheight" here and must use
"fontheight" instead.
  height = portal[details].fontheight * 3

 ~if this is a "sizing" calculation, we're done
  if (issizing <> 0) then
    done
    endif

 ~position our tallest portal at the top
  portal[details].top = 0

 ~determine whether the original name needs to be shown (only
if renamed)
  portal[original].visible =  !field[username].isempty

 ~center the info and delete portals within the upper half of the
template
  var half as number
  half = height / 2
  portal[info].top = (half - portal[info].height) / 2
  portal[delete].top = (half - portal[delete].height) / 2

 ~center the trappings portal within the lower half of the template
  ~Note: The info portal is big, so shift it down a little extra for
better spacing.
  portal[trappings].top = half + (half - portal[trappings].height) / 2
+ 1

 ~center the name vertically if we don't need the original name;
otherwise,
  ~position the two names above and below each other
  if (portal[original].visible = 0) then
    perform portal[name].centervert
  else
    portal[name].top = (half - portal[name].height) / 2 + 3
    perform portal[original].alignrel[ttob,name,2]
    endif

 ~position the delete portal on the far right
  perform portal[delete].alignedge[right,0]

 ~position the info portal to the left of the delete button
  perform portal[info].alignrel[rtol,delete,-6]

 ~center the trappings portal between the info and delete buttons
  var span as number
  span = portal[delete].right - portal[info].left
  portal[trappings].left = portal[info].left + (span -
portal[trappings].width) / 2

 ~position the name portals on the left and use a suitable
amount of space
  portal[name].left = 0  portal[name].width = 175
  portal[original].left = 15
  portal[original].width = portal[name].width - portal[original].left

 ~position the details next to the name and use the remaining
space
  portal[details].width = portal[info].left - portal[details].left - 10

 ~shrink the name portals if they don't fit, then re-align after the
shrink
  perform portal[name].sizetofit[32]
  if (portal[original].visible = 0) then
    perform portal[name].centervert
  else
    perform portal[original].sizetofit[32]
    portal[name].top = (half - portal[name].height) / 2 + 3
    perform portal[original].alignrel[ttob,name,2]
    endif
```

Revise Arcane Power Display

The trappings for an arcane power are noticeably missing from the display for each power on the "Arcane" tab. The original reason for this was that we didn't have the space, which was true at the time. However, we've now added the smarts to shrink the name when necessary, plus we've reduced the space for the name a little bit. If needed, we can reduce that space a bit further. We can also reduce the font size slightly for the various facets of each power, which should give us adequate room to fit both the point cost and range on the top line. This will free up the space to add a third line at the bottom that shows the trappings for the power.

The most obvious solution would be to revise the Label script for the "details" portal to put the range on the same line as the cost. This works great, except that we have no way of handling the situation where the range extends past the right edge. Since we are using a multi-line label portal, if the text extends past the edge on one line, that line is simply wrapped to the new one. What we need is for the right edge to simply be clipped if it is too long. The only way to handle that is by using a single-line label.

This means that we need to carve up our current "details" portal into two separate portals - one for the cost and range, plus another for the duration. We're also going to need to add a third portal for the trappings. Once we add another portal for the trappings, we'll have two portals conveying the same information, so we really should distinguish them clearly. It makes more sense to rename the current "trappings" portal to "trapedit" to indicate its use for editing the trappings, so we'll do that first.

After completing the rename operation, we can then add our two new portals. We can clone the current "details" portal to create a "duration" portal and a "trappings" portal. The Label script for each can be modified easily to output the appropriate information. We also need to modify the "details" portal to eliminate the duration and move the range up to the same line, leaving a horizontal gap between the two. Don't forget that all of these portals must be changed to be single-line portals, so remove the "ismultiline" attribute from each. This results in the three portals shown below.

```
<portal
  id="details"
  style="lblSmlLeft">
  <label>
    <labeltext><![CDATA[
      @text = "{/b}Points:{b} " & field[powPoints].text
      @text &= "{horz 20}{/b}Range:{b} " & field[powRange].text
      ]]></labeltext>
    </label>
  </portal>

<portal
  id="duration"
```

```
    style="lblSmlLeft">
    <label>
      <labeltext><![CDATA[
        @text = "{/b}Duration:{b} " & field[powLength].text
        if (field[powMaint].isempty = 0) then
          @text &= " (" & field[powMaint].text & ")"
          endif
        ]]></labeltext>
      </label>
    </portal>

  <portal
    id="trappings"
    style="lblSmlLeft">
    <label>
      <labeltext><![CDATA[
        @text = "{/b}Trappings:{b} " & field[powTraps].text
        ]]></labeltext>
      </label>
    </portal>
```

We now need to integrate these portals into the positioning logic for the template. Our total height of the template is now the combined height of the three portals. We need to position these portals beneath each other along the vertical axis, and they need to span the same region horizontally. Making these changes yields a revised Position script that looks like the following.

```
  ~set up our height based on our tallest vertical span, which is
the three
   ~portals for the power details
   height = portal[details].height + portal[duration].height +
portal[trappings].height

  ~if this is a "sizing" calculation, we're done
  if (issizing <> 0) then
    done
    endif

  ~determine whether the original name needs to be shown (only
if renamed)
   portal[original].visible =  !field[username].isempty

  ~center the info and delete portals within the upper half of the
template
   var half as number
   half = height / 2  portal[info].top = (half - portal[info].height) / 2
   portal[delete].top = (half - portal[delete].height) / 2

  ~center the trappings edit portal within the lower half of the
template
   ~Note: The info portal is big, so shift it down a little extra for
better spacing.
   portal[trapedit].top = half + (half - portal[trapedit].height) / 2 + 1

  ~center the name vertically if we don't need the original name;
otherwise,
   ~position the two names above and below each other
   if (portal[original].visible = 0) then
     perform portal[name].centervert
   else
     portal[name].top = (half - portal[name].height) / 2 + 3
     perform portal[original].alignrel[ttob,name,2]
     endif

  ~position the power details portals vertically
   portal[details].top = 0
   perform portal[duration].alignrel[ttob,details,0]
   perform portal[trappings].alignrel[ttob,duration,0]

  ~position the delete portal on the far right
   perform portal[delete].alignedge[right,0]
  ~position the info portal to the left of the delete button
   perform portal[info].alignrel[rtol,delete,-6]

  ~center the trappings edit portal between the info and delete
buttons
   var span as number
   span = portal[delete].right - portal[info].left
   portal[trapedit].left = portal[info].left + (span -
portal[trapedit].width) / 2

  ~position the name portals on the left and use a suitable
amount of space
   portal[name].left = 0
   portal[name].width = 175
   portal[original].left = 15
   portal[original].width = portal[name].width - portal[original].left

  ~position the details portals next to the name and use the
remaining space
   perform portal[details].alignrel[ltor,name,10]
   portal[details].width = portal[info].left - portal[details].left – 10
   portal[duration].left = portal[details].left
   portal[duration].width = portal[details].width
   portal[trappings].left = portal[details].left
   portal[trappings].width = portal[details].width

  ~shrink the name portals if they don't fit, then re-align after the
shrink
   perform portal[name].sizetofit[32]
   if (portal[original].visible = 0) then
     perform portal[name].centervert
   else
     perform portal[original].sizetofit[32]
     portal[name].top = (half - portal[name].height) / 2 + 3
     perform portal[original].alignrel[ttob,name,2]
     endif
```

This is looking pretty good, but the various details portals look a bit too large and cramped. It would look better if we shrank the font size a little bit. This will require that we define a new style. Looking at the style and font that are currently used, we can create a new style that has similar characteristics but with a smaller font. The new style should look like the one below.

```
<style
  id="lblSmPower">
  <style_label
    textcolor="f0f0f0"
    font="fntsmpower"
    alignment="left">
    </style_label>
  <resource
    id="fntsmpower">
    <font
      face="Arial"
      size="34">
      </font>
    </resource>
  </style>
```

After we switch our three portals over to using this new style, the display looks significantly better. The one thing that looks poor is that the default trappings almost always run off the end. Since those trappings are just to provide suggestions to a player, we really shouldn't even be showing them here. What we should ideally be showing is a message that tells the user to specify the appropriate trappings by clicking on the button at the right. We already treat unchanged trappings as a validation error, so we would be consistent if we displayed the message in red. This change can be

easily made within the Label script for the "trappings" portal, which will end up looking like the code below.

```
@text = "{/b}Trappings:{b} "
 if (field[powTraps].ischanged = 0) then
   @text &= "{text ff0000}Use button at right to specify"
 else
   @text &= field[powTraps].text
   endif
```

That's all there is to it. We've now got arcane powers working very smoothly.

Show Gear in Two Columns

The table on gear on the "Gear" tab uses a single column. The vast majority of gear have names that are relatively short, which results in a large gaps of empty space for each item in the table. This requires users to scroll through the list to see and access all their gear. Switching to a two-column table would work much better.

The first step is to change the "grGear" table portal to show only two columns. Now we need to revise the "grGrPick" template to get everything to fit better with the new design. We'll start by shrinking the horizontal margin down to two. Next, we'll shrink the gap between the buttons on the right from eight to two pixels. This is a big help, but some names are still too long. So the final step is to add code to shrink the name if it is too big, after which we must re-center the name vertically. The various effected lines of the Position script are shown below.

```
~position the info portal to the left of the delete button
 perform portal[info].alignrel[rtol,delete,-2]

~position the gear portal to the left of the info button
 perform portal[gearmanage].alignrel[rtol,info,-2]

 ....

~shrink the name to fit the available space if it's too big, then re-
center
 perform portal[name].sizetofit[30]
 perform portal[name].centervert
```

Showing Power for Weird Science

The "static" form at the top shows the power points for a character with an arcane background. This includes both the total points and the current points. However, if the character possesses the "Weird Science" background, each gizmo has its own points and the character itself has only a total number of points. So we must change the display to only show the total points for such as character.

The information shown within the form is pulled from the "acPPSumm" field on the actor. This field synthesizes the displayed text via a Finalize script. We can change that script to detect a character with "Weird Science" and adjust the result accordingly. The revised script should look like the one below.

```
~if the character uses weird science, only show total power
points
 @text = ""
 if (hero.tagis[Arcane.WeirdSci] = 0) then
   @text = hero.child[trkPower].field[trkLeft].value & " / "
   endif
@text &= hero.child[trkPower].field[trkMax].value
```

Weird Science Gizmos (Savage)

We're going to dedicate a full section to this topic. The actual changes are not complex, but there are a variety of issues that need to be considered. This results in a series of steps that we'll address one at a time.

How Gizmos Differ

Gizmos are a special variety of arcane power. In addition to the power itself, gizmos need to be controlled. In general, the use of a gizmo entails use of the "Weird Science" skill. However, a "ray gun" gizmo might use the "Shooting" skill instead. We need to track the associated skill for each gizmo and allow the user to specify that skill.

Gizmos are devices, so they each have their own pool of power points. This pool needs to be tracked independently for each gizmo. That means we need to integrate the proper handling for trackers into gizmos and display them on the "In-Play" tab so that user can manage them.

The final wrinkle with gizmos is that they can be shared. This means that a character with no arcane background can possess a gizmo. We therefore have to allow all characters to possess gizmos, and gizmos from other characters must allow the player to specify the number of power points available to each gizmo. We even have to handle the case where a weird scientist has both his own gizmos and gizmos created by others.

Differentiating a Gizmo

The first thing we need to resolve is how we're going to differentiate a gizmo from a normal power. One option would be to handle gizmos and powers much like we do weapons. We could define a new "base" component that has all the common behaviors of the two, then have separate "Power" and "Gizmo" components with the distinct behaviors. This would entail adding a new component set for gizmos as well. While this approach would definitely work, it's a non-trivial amount of work, so we should only go that route if it's truly necessary.

Gizmos are a proper superset of powers (i.e. they add behaviors but don't change any behaviors). In addition, there are only two pieces of new information that need to be tracked for powers. This means that the various new components we're considering would each be relatively small. An easier solution might be to just add the gizmo behaviors to the "Power" component and then use a tag to indicate whether a power is actually a gizmo. Since the changes we need to make aren't very complicated, using a tag ought to be more than adequate, so we'll give this approach a try.

We're actually going to have two different types of gizmos when we're finished. We need to identify gizmos in general, which can be handled via one tag. We also need to differentiate between gizmos created by the character and gizmos borrowed from another character. We'll refer to borrowed gizmos as "foreign" gizmos, and we can handle this distinction via a second tag. For simplicity, we'll just define these tags within the "Helper" tag group, and the new tags should look like below.

```
<value id="Foreign"/>
 <value id="Gizmo"/>
```

We'll worry about how and when to assign these tags in the next section.

MANAGING SHARED GIZMOS

Gizmos that are shared need to be managed appropriately by the user. We currently only show the "Arcane" tab when a character has an arcane background, but we could always change that behavior. If we always showed the "Arcane" tab, then shared gizmos could be handled on it. The problem with this approach is that shared gizmos are more generally perceived as "gear" - instead of arcane powers by - by characters that are using them. As such, it would probably be better if we added shared gizmos to the "Gear" tab.

Our first task is to decide how we're going to handle shared gizmos on the separate tab. We can easily share the same template for displaying shared gizmos and powers. The question is whether we can also share the same table portal. Unfortunately, we can't. The two tables need to show different collections of powers/gizmos, so we need to use a different List tag expression on each. The existing table portal on the "Arcane" tab needs to show powers that do not possess the "Helper.Foreign" tag, which the table on the "Gear" tab only shows powers that do possess the tag. We also want to change the various text displayed for the header, adding an item, etc.

We can modify the "apPowers" table portal on the "Arcane" tab to have a suitable List tag expression, yielding the portal shown below.

```
<portal
  id="apPowers"
  style="tblNormal">
  <table_dynamic
    component="Power"
    showtemplate="apPower"
    choosetemplate="SimpleItem"
    addpick="resPowers"
    descwidth="350">
    <list>!Helper.Foreign</list>
    <titlebar><![CDATA[
      @text = "Add an Arcane Power - " &
hero.child[resPowers].field[resSummary].text
      ]]></titlebar>
    <description/>
    <headertitle><![CDATA[
      @text = "Arcane Powers -  " &
hero.child[resPowers].field[resSummary].text
      ]]></headertitle>
    <additem><![CDATA[
      ~get the color-highlighted "add" text
      @text = field[resAddItem].text
      ]]></additem>
    </table_dynamic>
  </portal>
```

For our new table portal on the "Gear" tab, we can start by copying the "apPowers" portal. We can then give it a new id, change the List tag expression to only include powers that possess the tag, and revise the various text shown via the scripts. However, there is one important question we haven't resolved yet. How do we get the "Helper.Foreign" tag to be assigned to items added via our new table?

The answer to this is the "autotag" element. Table portals can assign tags to each pick that they add to the hero. The assigned tags behave as if they are part of the defined nature of the pick. Consequently, by specifying an "autotag" element that assigns the "Helper.Foreign" tag, all picks added by the new table possess the tag, while all picks added by the original table on the "Arcane" tab do not.

Putting all this together results in the following new table portal being defined.

```
<portal
  id="grGizmos"
  style="tblNormal">
  <table_dynamic
    component="Power"
    showtemplate="apPower"
    choosetemplate="SimpleItem"
    descwidth="350">
    <list>Helper.Foreign</list>
    <autotag group="Helper" tag="Foreign"/>
    <titlebar><![CDATA[
      @text = "Add a Borrowed Weird Science Gizmo"
      ]]></titlebar>
    <description/>
    <headertitle><![CDATA[
      @text = "Borrowed Weird Science Gizmos"
      ]]></headertitle>
    <additem><![CDATA[
      @text = "Add a Borrowed Weird Science Gizmo"
      ]]></additem>
    </table_dynamic>
  </portal>
```

The final thing we need to do is integrate our new table portal into the layout. The layout for gear currently contains all the gear and any vehicles. We'll insert the shared gizmos between the two tables. We can use the same approach as is already used for the vehicles, reserving space for a single item in the table. This way, we can easily integrate the new portal and dedicate the majority of space on the table to normal gear. If there is space left over, we'll allocate it first to additional shared gizmos and lastly to additional vehicles. The revised layout should look like the one below.

```
<layout
  id="gear">
  <portalref portal="grGear" taborder="10"/>
  <portalref portal="grGizmos" taborder="20"/>
  <portalref portal="grVehicle" taborder="30"/>

  <!-- This script sizes and positions the layout and its child
visual elements. -->
  <position><![CDATA[
    ~set all tables to span the full width of the layout
    portal[grGear].width = width
    portal[grGizmos].width = width
    portal[grVehicle].width = width

    ~position the vehicles table at the bottom with a minimum
height of 2 items
    portal[grVehicle].maxrows = 2
    portal[grVehicle].top = height - portal[grVehicle].height

    ~position the gizmos table above the vehicles table with a
minimum of 1 item
    portal[grGizmos].maxrows = 1
    portal[grGizmos].top = portal[grVehicle].top -
portal[grGizmos].height

    ~position and size the gear table to fill all remaining space
    portal[grGear].top = 0
    portal[grGear].height = portal[grGizmos].top - 10

    ~position and size the gizmos and vehicle tables to use the
remaining space
    ~at the bottom
    portal[grGizmos].top = portal[grGear].bottom + 10
    portal[grGizmos].height = portal[grVehicle].top -
portal[grGizmos].top
    portal[grVehicle].top = portal[grGizmos].bottom + 10
```

```
    portal[grVehicle].height = height - portal[grVehicle].top
    ]]></position>

  </layout>
```

If we reload the data files and give our new table a try, we discover that we have a problem. No matter what we do, HL tells us that there are no powers to choose from. But we know that there are powers available, because we can readily add those powers on the "Arcane" tab. The problem is due to our List tag expression, which only includes powers that possess the "Helper.Foreign" tag. By default, the Kit assumes that the restrictions set forth by the List tag expression should also apply when identifying items for selection. However, the "Helper.Foreign" tag is only added when things are added via the table, so none exist with the tag pre-assigned, hence there is nothing to select.

The solution is to add our own Candidate tag expression, which tells the Kit to ignore the List tag expression by default. The Candidate tag expression doesn't actually have to contain anything, since we don't need to perform any special candidate tests - it must only exist. So all we need to do to fix the problem is add an empty Candidate tag expression to our table portal, which consists of the XML element below.

```
<candidate/>
```

If we reload the data files and try again, we now have no problems selecting powers from our new table. More importantly, powers added via our new table only appear within that table, and powers added on the "Arcane" tab only appear within that table. This means that a character with an arcane background can also borrow gizmos from other characters, with the two groups being fully distinguished.

ADDING THE FIELDS

We can now begin adding the logic for gizmos to powers. There are two separate pieces of information that we need to maintain for gizmos. The first is the linked skill and the second is the power points for the gizmo. We'll need one new field for each of these bits of information.

The linked skill will need to be selected by the user from a list of all the various skills. As such, we'll need to utilize a menu for the skill. Menu selections require special handling for the fields in which the selection will be saved. We must properly designate the field as being used for saving a menu selection by assigning it the correct "style". The resulting field definition is shown below.

```
<field
  id="powSkill"
  name="Linked Skill"
  type="user"
  style="menu">
  </field>
```

The stored power points for a gizmo are simpler. It's a value-based field that can be modified by the user. This yields the following field definition.

```
<field
  id="powStorage"
  name="Stored Points"
  type="user">
  </field>
```

Now that we've added the fields, we should also determine when we'll be using them. This will be dictated by the presence of the "Helper.Gizmo" and "Helper.Foreign" tags. The latter tag will be handled exclusively via the table portal itself, with shared gizmos receiving the tag. However, the "Helper.Gizmo" tag needs to be assigned correctly, so we'll add an Eval script to the component for that purpose. There are two situations when we need to assign the tag. The first is when the power has the "Helper.Foreign" tag, since we know the power is intended for use as a shared gizmo. The second is when the character possesses the "Weird Science" arcane background, since all of his powers must be treated as gizmos. This results in the Eval script below.

```
<eval index="3" phase="Setup" priority="5000"><![CDATA[
  ~it's a gizmo if it's foreign or if the character is a weird
scientist
  if (tagis[Helper.Foreign] + hero.tagis[Arcane.WeirdSci] <> 0)
then
    perform assign[Helper.Gizmo]
    endif
  ]]></eval>
```

REVISING THE TEMPLATE

Arcane powers for a character with the "Weird Science" arcane background must be treated as gizmos on the "Arcane" tab. So the same template will be used for both arcane powers and gizmos. We're also going to be using this template to assign the power points for shared gizmos on the "Gear" tab.

The first thing we need to do is decide how we're going to visually integrate the new fields for gizmos into the template. We'll need a menu and an edit portal for the linked skill and the power points, plus two labels to identify them. The best solution is probably to add the gizmo-related fields at the bottom, beneath the trappings display. This way, a gizmo looks identical to a non-gizmo, except for the additional information at the bottom. This approach also make things easiest to implement.

We'll now define the four new portals we need. Since they will be positioned beneath the trappings, we'll insert them into the template right after the "trappings" portal. This way, we'll get the desired order when the user employs the <Tab> key to move between portals. For the menu, we'll specify that we want to select skills and that the default selection should be the "Weird Science" skill. We also need to specify the selection of "things" instead of "picks", since shared gizmos may be used by characters without the requisite skill (e.g. "Weird Science"). For the edit portal, we'll specify an integer format with a maximum of two digits. This yields the portals shown below.

```
<portal
  id="lblmenu"
  style="lblSmPower">
  <label
    text="Skill:">
    </label>
  </portal>

<portal
  id="menu"
  style="menuSmall">
  <menu_things
    field="powSkill"
```

```
    component="Skill"
    maxvisible="10"
    defthing="skWeirdSci"
    usepicks="thing">
    </menu_things>
  </portal>

<portal
  id="lblstorage"
  style="lblSmPower">
  <label
    text="Points:">
    </label>
  </portal>

<portal
  id="storage"
  style="editNormal">
  <edit
    field="powStorage"
    format="integer"
    maxlength="2">
    </edit>
  </portal>
```

Now that the portals are defined, we can manage them appropriately within the Position script. Since the visibility of the gizmo-related portals will influence the total height of the template, we need to determine their visibility first. The menu portal and its associated label are only shown for gizmos. The power points edit portal and its associated label are only shown for foreign gizmos. This results in the new script code below being inserted at the top.

```
~determine if the menu portal and label should be visible or not
if (tagis[Helper.Gizmo] = 0) then
  portal[lblmenu].visible = 0
  portal[menu].visible = 0
  endif

~only show the power points edit field if this is a foreign gizmo
 if (tagis[Helper.Foreign] = 0) then
  portal[lblstorage].visible = 0
  portal[storage].visible = 0
  endif
```

The menu portal will definitely be taller than the edit portal and either of the labels, so its height will be factored into the overall height of the template. If the menu portal is visible, we need to increase the height of the template accordingly. We should also insert a tiny gap between the menu and the trappings details above - one pixel should suffice. The revised code for calculating the height should look like below.

```
 ~set up our height based on our tallest vertical span, which is
the three
  ~portals for the power details, plus the menu if visible
  height = portal[details].height + portal[duration].height +
portal[trappings].height
  if (portal[menu].visible <> 0) then
    height += portal[menu].height + 1
    endif
```

After we position the various details portals for the power, we can position the gizmo portals beneath them. Since the menu portal is the tallest, we'll position it first and then center the other portals upon it. This results in the code below, which can be inserted immediately after the vertical positioning of the power details portals.

```
~position the gizmo-related portals vertically
 perform portal[menu].alignrel[ttob,trappings,1]
 perform portal[lblmenu].centeron[vert,menu]
 perform portal[lblstorage].centeron[vert,menu]
 perform portal[storage].centeron[vert,menu]
```

The final task is to position the gizmo-related portals horizontally. We must wait to do that until after we've positioned the details portals horizontally, since we need to utilize the same horizontal span. Within that span, we position the portals sequentially, with appropriate spacing. We also need to decide upon appropriate widths for the menu and edit portal. The result is the code below, which can be inserted after the details portals are positioned horizontally.

```
~position the gizmo-related portals horizontally
 portal[lblmenu].left = portal[details].left
 perform portal[menu].alignrel[ltor,lblmenu,3]
 portal[menu].width = 135
 perform portal[lblstorage].alignrel[ltor,menu,10]
 perform portal[storage].alignrel[ltor,lblstorage,2]
 portal[storage].width = 25
```

Our new portals are positioned properly and appear only when appropriate.

Tracking Power Points

Gizmos are now appearing as they should within the tables, but they still aren't behaving correctly. Each gizmo needs to appear in the list of trackers on the "In-Play" tab. This requires that we make every power incorporate all the behaviors of a tracker. To do that, we must modify the "Power" component set to include the "Tracker" component, which yields the revised component set below.

```
<compset
  id="Power">
  <compref component="Power"/>
  <compref component="MinRank"/>
  <compref component="Tracker"/>
  </compset>
```

Powers now possess all the behaviors of trackers, but we need to configure them properly. If we start experimenting, the first thing we'll notice is that every power now shows up in the list of trackers on the "In-Play" tab. We only want gizmos to show up in that list, since powers don't actually have a separate set of points. This means that every power that is not a gizmo must be assigned the "Hide.Tracker" tag. We already added an Eval script that determines when we have a gizmo, so we can simply modify it to assign the tag for non-gizmos. The revised Eval script looks like the following.

```
 <eval index="3" phase="Setup" priority="5000"><![CDATA[
   ~it's a gizmo if it's foreign or if the character is a weird
scientist
   if (tagis[Helper.Foreign] + hero.tagis[Arcane.WeirdSci] <> 0)
then
     perform assign[Helper.Gizmo]

  ~if not a gizmo, hide the power from the list of trackers on in-
play tab
   else
    perform assign[Hide.Tracker]
    endif
  ]]></eval>
```

The next thing we need to do is setup each gizmo to possess the proper number of power points. The actual number of power points used by the tracker is pulled from the "trkMax" field. So we need to make sure that this field is setup correctly. We'll define a new Eval script for this purpose. In the script, we'll pull the power points for a foreign gizmo from the "powStorage" field that the user will specify. For a gizmo belonging to a character with "Weird Science", we need to use the character's maximum power, which can be obtained from the "trkPower" tracker. Since the value we set must be in place before the tracker calculates the amount remaining, we must be sure to schedule our new script before the calculation occurs. This yields the new Eval script below, which we'll define for the component.

```
<eval index="4" phase="Traits" priority="10000">
  <before name="Calc trkLeft"/><![CDATA[
  ~if we're not a gizmo, there's nothing to do
  if (tagis[Helper.Gizmo] = 0) then
    done
    endif

  ~if the power is not foreign, setup its power points
  if (tagis[Helper.Foreign] = 0) then
    field[trkMax].value = #trkmax[trkPower]

  ~otherwise, setup the power points for a foreign gizmo
  else
    field[trkMax].value = field[powStorage].value
    endif
  ]]></eval>
```

We can now test out how gizmos are working. The trackers show up properly for each gizmo, and the total points for each gizmo reflect the proper values. The only problem is that resetting a gizmo on the "In-Play" tab sets the value to zero. When a gizmo is reset, it should be assigned the maximum. We can fix this by changing the behavior of the tracker, which requires assigning the "Helper.ResetMax" tag. Since we need this behavior on all gizmos, we can assign it to the "Power" component with the line below.

```
<tag group="Helper" tag="ResetMax"/>
```

Gizmos are now working the way the should.

Validation

There are a couple things left that we have not done. Conditions exist that we should be reporting as an error to the user. If a user doesn't select a skill for a gizmo, we should report it as an error. Similarly, if the user doesn't specify a non-zero number of power points for a foreign gizmo, we should also report an error. We can resolve each of these by adding suitable Eval Rule scripts to the component, as shown below.

```
 <evalrule index="2" phase="Validate" priority="6000"
message="Linked skill must be specified for gizmo"><![CDATA[
   if (tagis[Helper.Gizmo] = 0) then
     @valid = 1
   elseif (field[powSkill].ischosen <> 0) then
     @valid = 1
     endif
   ]]></evalrule>

 <evalrule index="3" phase="Validate" priority="7000"
message="Power points must be specified for
gizmo"><![CDATA[
   if (tagis[Helper.Gizmo] = 0) then
     elseif (tagis[Helper.Foreign] = 0) then
       @valid = 1
     elseif (field[powStorage].value <> 0) then
       @valid = 1
       endif
     ]]></evalrule>
```

Since failing to select a linked skill is now considered a validation error, we should highlight such a condition to the user in red. If the menu is invalid, we can modify the menu style appear in red. Adding the code below at the end of the Position script will work nicely.

```
~if the menu is visible and nothing is chosen yet, flag it in red
 if (portal[menu].visible <> 0) then
   if (field[powSkill].ischosen = 0) then
     perform portal[menu].setstyle[menuError]
     endif
   endif
```

Dashboard (Savage)

All of the critical mechanics for individual characters are now in place. It's time to turn our attention to managing multiple characters effectively. The first piece in that process is the Dashboard.

Determine the Display Content

The Skeleton files provide a basic framework for the Dashboard. The character name is shown at the top. Various mouse-over icons are arrayed across the bottom. Buttons to switching between actors and move gear are down the right side. We need to figure out what additional information we want to show to the user. We don't have much space, so we need to choose carefully. The most frequently referenced information should be given priority and shown.

Looking closely at the make-up of Savage Worlds characters, there are five basic pieces of information that seem most important. These are listed below.

- **Health Status:** The number of wounds and fatigue level are important, as is whether the character is shaken.
- **Parry:** In a game with lots of combat, the Parry trait will be referenced frequently.
- **Toughness:** The rationale for Toughness is similar to the Parry trait.
- **Power Points:** It's also valuable to know how many power points remain for a character with an arcane background.
- **Bennies:** Bennies are an important resource that need to be managed by the player, so knowing how many you have left at-a-glance can be extremely useful.

It's possible that we won't be able to squeeze all of this information into the limited space we have for each actor on the Dashboard. So we need to assess each one and determine both how to display it and where it might fit. At the end, if don't have room, we need to drop the lower priority piece(s) of information.

Health is probably the most important piece, although it also requires significant space. If a character is wounded, fatigued, and shaken, all three facets must be indicated in some way. We already use a very compact representation for this within the static form at the top of the main window, but it still consumes a fair bit of space. If we can't think of a more compact representation, we'll be hard-pressed to show all the other information on the Dashboard. The primary use for the Health as a quick-reference during play is to

show the net impact on the character. As such, we could display a shaken indicator and only the total negative adjustment for the character. This would cut the required space in half.

Since combat tends to factor prominently in many Savage Worlds games, the Parry and Toughness traits are probably the next most important. These traits are simple values, so they don't require much space to display on the Dashboard. The Power Points are likely more important than the Bennies. Showing the Power Points can be done in two different ways. One option is to show them as we do on the static form at the top, with both the current and maximum values, but we could also just show the current value for space efficiency. The same two options apply for Bennies.

Let's consider how and where we can squeeze everything in now. The default behavior for the Dashboard is to reserve two lines of space for the actor name. The primary reason for this is to be able to show the associations for minions and masters. Since allies often play a significant role in Savage Worlds games, and since they can best be modeled as minions of the PCs, it would be nice to retain this space on the Dashboard.

If we retain that space, though, we're left with some hard decisions. Beneath the name, there really isn't enough space to show more than one line of information. Based on our prioritization above, this leaves us with the character's Health (in a compressed format) and room for probably only two traits (Parry and Toughness). We'll go with this design for now and see if there's a way we can squeeze in the Power Points at the end.

New Health Format

In the previous section, we decided on a more compact format for showing Health on the Dashboard. Now we need to synthesize that format appropriately. The current format is generated through a field on the "Actor" component. Probably the best method for us to generate the new format is via an alternate field on the same component.

We can start by cloning the existing "acDmgSumm" field and giving it a new id. We'll use "acDmgTac", since we'll be using this format on both the Tactical Console and Dashboard. Then we can revise the Finalize script to correctly synthesize the new format. The resulting new field is shown below.

```
<field
  id="acDmgTac"
  name="Health for Dashboard/TacCon"
  type="derived"
  maxfinal="100">
  <!-- Calculate a value that is based on all the fields referenced
by the
       "finalize" script. This ensures that the field always
changes if any of
       its pieces changes, which triggers the finalized value to
be updated.
  -->
  <calculate phase="Render" priority="1000"><![CDATA[
    ~make sure this value consists of the elements that could
cause the summary to change
    @value = field[acShaken].value * 10000 +
field[acWounds].value * 100 + field[acFatigue].value
    ]]></calculate>
  <!-- Final value for display shows the shaken state, current
wounds, and any penalty value
.  -->
  <finalize><![CDATA[
    ~if we're not shaken and have incurred no negative effects,
all is good
    net = field[acWounds].value + field[acFatigue].value
    if (field[acShaken].value + net = 0) then
      @text = "-"
      done
      endif

    ~if we're shaken, signal it with a special indicator
    @text = ""
    if (field[acShaken].value <> 0) then
      @text &= "{font wingdings}v{revert}"
      endif

    ~get the current wounds and fatigue for use below
    var wounds as number
    var fatigue as number
    wounds = field[acWounds].value
    fatigue = field[acFatigue].value

    ~if we're incapacitated, report it
    var state as string
    if (wounds >= 4) then state = "Inc"
    elseif (fatigue >= 3) then
      state = "Inc"

    ~otherwise, report our total negative influence
    elseif (net > 0) then
      state = "-" & net
      endif

    ~if we have a state to report, append it
    if (empty(state) = 0) then
      if (empty(@text) = 0) then
        @text &= "/"
        endif
      @text &= state
      endif

    ~anything we output must be in red to highlight the penalty
    @text = "{text ff0000}" & @text
    ]]></finalize>
</field>
```

Implement the Display

It's now time to implement the changes to the Dashboard. Open the file "form_dashboard.dat" and locate the "dashboard" template. The first thing we need to do is integrate our new field for showing the Health. The "health" portal already exists, so all we need to do is change the field it displays to the new "acDmgTac" field. This results in the revised portal below.

```
<portal
  id="health"
  style="lblSmall">
  <label>
    <labeltext><![CDATA[
      @text = field[acDmgTac].text
      ]]></labeltext>
    </label>
  </portal>
```

The "power" portal is being eliminated, but we'll want to do something similar for showing the "Parry" and "Toughness" traits. We can adapt the portal for the "Parry" trait, then we can clone it and revise it for the "Toughness" trait. The following two portals should result, along with the effective deletion of the "power" portal.

```
<portal
  id="parry"
```

```
    style="lblSmall">
    <label>
      <labeltext><![CDATA[
        @text = "{size 30}Pr: {size 36}" & #trait[trParry]
        ]]></labeltext>
      </label>
    </portal>

  <portal
    id="toughness"
    style="lblSmall">
    <label>
      <labeltext><![CDATA[
        @text = "{size 30}To: {size 36}" & #trait[trTough]
        ]]></labeltext>
      </label>
    </portal>
```

The portals are now in place, so we need to position everything properly. We'll position the "toughness" portal adjacent to the "gear" portal, then we'll place the "parry" portal on the left of that. The "health" portal will then be influenced by the "parry" portal. We can also introduce a little bit more spacing to make things look good. This results in the following revised script code for positioning the three portals.

```
~position toughness details next to the "gear" portal
 perform portal[toughness].centeron[vert,gear]
 perform portal[toughness].alignrel[rtol,gear,-6]

~position parry details next to the "toughness" portal
 perform portal[parry].centeron[vert,toughness]
 perform portal[parry].alignrel[rtol,toughness,-6]

~position health details parallel to the "parry" portal
 perform portal[health].centeron[vert,parry]
 portal[health].width = portal[parry].left - 4
```

The final thing we need to do is handle the vertical adjustment to our new portals if the name requires two lines. The revised script code should look as follows.

```
 ~shift the health and related portals down slightly to make more
room
  portal[health].top += 2
  portal[parry].top += 2
  portal[toughness].top += 2
```

If we reload the data files and take a look at the Dashboard, it looks reasonable. Unfortunately, there just isn't any space to squeeze in the Power Points, so we'll just have to leave them out.

Mouse-Over Information

The final task associated with the Dashboard is to generate suitable mouse-over text for each of the five icons across the bottom. The Skeleton files utilize five different procedures for this purpose, and they are already hooked into the appropriate portals. All we need to do is revise the individual procedures to show the information we want.

Special Abilities

Moving from left to right, the first mouse-over icon is intended to show all of the special abilities for the actor. The list presented should generally contain the same items as are shown within the "Special" tab. Looking at the "DshSpecial" procedure, we'll see the that all picks with the "DashTacCon.Special" tag are included in the list that is output. If we look at the "SpecialTab" component, we'll see that every thing derived from the component is assigned that tag. Consequently, everything that appears on the "Special" tab will also be shown by this procedure.

The question we need to ask ourselves is if there is any additional material that we want to include here, or if there is any material that we want to exclude. If there is, then we simply need to make sure to assign or delete the "DashTacCon.Special" tag for those things/picks. Adding the tag will output the new material the same way special abilities are output.

After giving it a bit of thought, there is one group of things we should add - arcane powers. However, we need to display arcane powers differently from the current special abilities. That means that we can't just assign them the tag. Instead, we need to integrate them into the logic of the procedure. The arcane powers should be kept visually separate from the special abilities. For each power, we should so the point cost, range, duration, and maintenance. Within the procedure, we can insert the code below just before the last line of the script.

```
 ~if we have any arcane powers, separate them from the special
abilities above
  if (hero.haschild["component.Power"] <> 0) then
    final &= "{br}{horz 10}{b}{u}Arcane Powers and/or Gizmos{/u}"
    final &= "  (" & herofield[acPPSumm].text & "){/b}{br}{vert 5}"
    endif

 ~output all arcane powers  foreach pick in hero where
"component.Power"
    final &= "{b}" & eachpick.field[name].text & "{/b} - "
    final &= "Pts: {b}" & eachpick.field[powPoints].text & "{/b}"
    final &= "; Rng: {b}" & eachpick.field[powRange].text & "{/b}"
    final &= "; Dur: {b}" & eachpick.field[powLength].text & "{/b}"
    if (eachpick.field[powMaint].isempty = 0) then
      final &= " (" & eachpick.field[powMaint].text & ")"
      endif
    final &= "{br}"
    nexteach
```

Skills and Other Rolls

Continuing to the right, the next mouse-over icon is intended to display the various rolls that apply for the actor. This includes skills and anything else that is used as a roll during play. We'll start by looking at the "DshRolls" procedure, where we find that it includes all picks with the "DashTacCon.Rolls" tag. This tag is included on every skill by the "Skill" component, but it is not assigned anywhere else.

There really aren't any other rolls that characters need to make, aside from attribute rolls. We could put the attributes here, but they are already included in the fourth mouse-over, as part of the basic character information. So the question is whether we want to move them. It's probably best to leave them where they are, so there are no changes necessary for this mouse-over.

Weapons and Combat

The central mouse-over icon displays all the combat-related details for the actor. Weapons, armor, shields, and combat-centric traits should be included here. The contents are governed by the "DshCombat" procedure, which outputs four groups of information. The first group consists of traits that are assigned the "DashTacCon.Combat" tag, followed by armor, shields, and weapons.

The traits to be included must be individually assigned the "DashTacCon.Combat" tag. So we'll start by assessing which traits should be included. The "Parry" and "Toughness" traits are obviously combat-related. The one questionable trait is "Pace". We'll go ahead and include for now. Once the list is determined, we can go through all traits and make sure that only those in our list possess the appropriate tag.

The next things output are any equipped armor and/or shield. For both, we need to include the defense rating. However, we need to include any parry bonus for shields and the areas covered for armor. We can revise the script code for these objects to reflect these changes, resulting in the following.

```
~output equipped armor and shield  var temp as string
info = ""  foreach pick in hero where "component.Armor"
  if (eachpick.field[grIsEquip].value <> 0) then
    info &= eachpick.field[name].text & " {b}" &
eachpick.field[defDefense].text & "{/b}"
    temp = eachpick.tagabbrevs[ArmorLoc.?,","]
    if (empty(temp) = 0) then
      info &= ", Covers: " & temp
      endif
    info &= "{br}"
    endif
  nexteach
if (empty(info) <> 0) then
  info = "-No Armor Equipped-{br}"
  endif
final &= info
info = ""
foreach pick in hero where "component.Shield"
  if (eachpick.field[grIsEquip].value <> 0) then
    info &= eachpick.field[name].text & " {b}" &
eachpick.field[defDefense].text & "{/b}"
    if (eachpick.field[defParry].value <> 0) then
      info &= ", Parry: {b}" &
signed(eachpick.field[defParry].value) & "{/b}"
      endif
    info &= "{br}"
    endif
  nexteach
if (empty(info) <> 0) then
  info = "-No Shield Equipped-{br}"
  endif
final &= info & "{br}"
```

The final block of information included is the weapons. Since the "Armory" sort set is utilized, equipped weapons will be shown first, whether they are melee weapons or ranged weapons. After that, ranged weapons are listed, followed by melee weapons. Within the code, we need to clean up how each weapon is displayed, as well as add details that are currently omitted (e.g. armor piercing, ranges, etc.). The revised code should look like below.

```
~output all weapons, with equipped ones first
info = ""
foreach pick in hero where "component.WeaponBase" sortas
Armory
  info &= eachpick.field[name].text & " {b}" &
eachpick.field[wpNetAtk].text & "{/b}"
  info &= ", Dmg: {b}" & eachpick.field[wpShowDmg].text &
"{/b}"
  if (eachpick.field[wpPiercing].value <> 0) then
    info &= ", AP" & eachpick.field[wpPiercing].text
    endif
  if (eachpick.tagis[component.WeapRange] <> 0) then
    info &= ", Rng: " & eachpick.field[wpRange].text
    endif
  if (eachpick.field[wpNotes].isempty = 0) then
    endif
  info &= "{br}"
  nexteach
if (empty(info) <> 0) then
  info = "-No Weapons-{br}"
  endif
final &= info
```

BASIC INFORMATION

The mouse-over for basic character information shows facets of the character that aren't included elsewhere. Details like race, attributes, and other derived traits belong here. The contents are controlled via the "DshBasics" procedure, so we'll look there first.

The race makes sense to include, but the power points need to be eliminated - they are already presented in the abilities mouse-over. The next block outputs "resistance" traits that are identified by the tag "User.Resistance". Savage Worlds doesn't have the notion of special resistance traits (e.g. saving throws), so we're not using any such picks. However, we do have a few traits that we need to show somewhere (e.g. Charisma). What we can do is define a new tag to identify the traits we want to included within the "Basics" mouse-over. We'll put it in the "DashTacCon" tag group and give it the id "Basics".

Once the tag is defined, we can assign it to the appropriate traits we want to include. The only trait that is not included anywhere is "Charisma", so we definitely need to assign it the tag. We should also probably assign the tag to the "Pace" trait. This trait is used outside of combat as well, and it really doesn't hurt to repeat it in both places, so we assign it the tag. Then we can revise the code to key on our new tag, as shown below.

```
~output our non-combat traits
 foreach pick in hero where "DashTacCon.Basics"
   final &= eachpick.field[name].text & ": {b}" &
eachpick.field[trtDisplay].text & "{/b}{br}"
   nexteach  final &= "{br}"
```

Attributes are shown next, which we definitely want to retain. Lastly, permanent adjustments are output, which we also want to retain. So our only changes to this mouse-over are the ones outlined above.

ACTIVE EFFECTS

The final mouse-over icon shows any temporary effects that are active on the character. This includes activated abilities and in-play adjustments, both of which are controlled via the "In-Play" tab. The contents are synthesized by the "DshActive" procedure, which requires no changes for our needs/

TACTICAL CONSOLE (SAVAGE)

With the Dashboard operating smoothly, we'll shift our focus to the Tactical Console, or TacCon for short. All the behaviors of the TacCon are orchestrated within the file "form_taccon.dat". In general, the only facet of the TacCon that you'll need to worry about as an author is the contents of the "tacPick" template, which controls the information shown for each individual actor. All other TacCon behaviors can be left in their default state, since those behaviors will be the same for just about every game system. This is true for Savage Worlds, so we'll only focus on the "tacPick" template below.

ORIENTATION

The contents of the "tacPick" template are organized into a number of regions. We'll start with a brief orientation to those regions, after which we'll begin revising each of those regions to suit our purposes.

Flanking the template on either side are character portraits. The image is placed on the left for actors that are allies of the PCs. Actors that are NPCs and/or established enemies of the PCs are have their portraits shown on the right. This provides an immediate visual cue for the user regarding ally versus enemy. As such, you should probably not change this behavior without good reason.

Moving from left to right, the next region of the template provides general information about the actor. This area is similar to the Dashboard, as it displays the actor's name and a small number of the most important details for the actor.

At the right edge of this region, two buttons are shown. One allows you to switch directly to an actor within the main window, just like is done on the Dashboard, while the other brings up a form where you can apply damage and other effects to the actor.

To the right of the buttons is a column of traits. The traits shown here are those that are most likely to be utilized by the user during play. This ensures that those traits are always visible for reference.

A large region in the center holds an assortment of details about the actor. The contents of this region will be different, depending on whether the actor is in combat or out of combat. During combat, equipped weapons and other combat-related details can be shown for quick reference by the user. When out of combat, other information can be shown, such as a summary of important skills that are commonly used.

In the upper right, a row of mouse-over icons is provided. These icons are identical to the ones utilized on the Dashboard. They are shown in the same order and are hooked up to the same procedures, so they will offer the same information to the user.

Beneath the mouse-over icons, status information about the actor is presented when not in combat.

Once combat begins, a number of changes take place. A region is reserved on the far left to indicate import state about the actor, such as whether a character has not yet acted in the combat. Next to this region, a set of three buttons is shown, providing control over the combat actions of the actor. The status information shown on the far right is moved to the line across the bottom of the template. Where the status was shown, an incrementer appears for the purpose of controlling initiatives. And at the far right edge, a set of buttons is shown for moving the actor up and down within the combat sequence.

BASIC ACTOR INFORMATION

The region on the left for basic actor information serves much the same purpose as on the Dashboard. The key difference is that the TacCon provides a bit more space. This means that we ought to be able to include all five pieces of information that we identified when working on the Dashboard. These pieces are the Health, Parry, Toughness, Power Points, and Bennies

There are basically two lines of space available to us. We'll use the same compact format for the Health that we created for the Dashboard. We'll put the Parry and Toughness traits on the same line with the Health. On the other line, we'll place the Power Points and Bennies, showing the current and maximum values for each.

There are two labels portals for displaying basic character information. They are named "status1" and "status2", with "status1" appearing above "status2". We'll put the Health on the top line. For the second line, we need to only show power points if the character actually possesses an arcane background. This results in the following two revised portals and Label scripts.

```
<portal
  id="status1"
 style="lblSmlLeft">
  <label
    ismultiline="yes">
    <labeltext><![CDATA[
      ~start with the health status
      @text = "{size 30}Hlth: {size 36}" & field[acDmgTac].text

      ~append the parry trait
      @text &= "{horz 10}{size 30}Pr: {size 36}" & #trait[trParry]

      ~append the toughness trait
      @text &= "{horz 6}{size 30}To: {size 36}" & #trait[trTough]
      ]]></labeltext>
    </label>
  </portal>

<portal
  id="status2"
  style="lblSmlLeft">
  <label
    ismultiline="yes">
    <labeltext><![CDATA[
      ~start with the power points status (if any)
      if (hero.tagis[Arcane.?] <> 0) then
        @text = "{size 30}PP: {size 36}" &
herofield[acPPSumm].text @text &= "{horz 12}"
        endif

      ~add the bennies
      @text &= "{size 30}Ben: {size 36}" &
hero.child[trkBennies].field[trkUser].text
      ]]></labeltext>
    </label>
  </portal>
```

IMPORTANT TRAITS

The column of traits to the right of the basic character information is managed via the "traits" portal. All of the traits that should appear in the list are identified by the "DashTacCon.Traits" tag. There is room to show up to three different traits in this portal. Since there are four derived traits in Savage Worlds, we need to decide which one to include in the list. Since we already show Parry and Toughness in the basic information region, we really don't need to duplicate them here, so we only include Pace and Charisma. Open the file "thing_traits.dat" and make sure that these two derived traits possess the tag.

NON-COMBAT CONTENT

The information shown in the center changes based on whether the party is in combat or not. When not in combat, the region should show useful facets of the character that a GM will refer to during play. For example, the character's skills, or perhaps just the most commonly used skills in a game where all characters possess a large number of skills.

In Savage Worlds, characters purchase their skills, so we can simply show all of the skills for each character. The region is controlled via the "peace" portal, which uses a Label script to synthesize its contents. All of the items shown in the region are identified by possessing the "DashTacCon.NonCombat" tag.

Since we copied our things when defining them, many of the standard skills are assigned the tag, while others are not. In addition, special skills like the arcane skills are not being assigned the tag, which means they are not included in the list. The arcane skills are just as important and should be displayed. We probably can't fit all skills in the available space, so we need to choose the skills to include. So let's go through all the skills and decide which ones should (or shouldn't) be included in the list. Be sure to check all skills, including those associated with arcane background and powers. Any skill to be shown must be assigned the tag.

Once we've identified the appropriate skills, we need to review the output being generated. Everything looks good, so our non-combat content is ready to go.

COMBAT CONTENT

Once combat begins, the skills shown in the center region are swapped out for information that is more appropriate during combat. More specifically, the first three weapons possessed by the character are shown, with any equipped weapon(s) being shown before unequipped weapons. For each weapon, the attack roll and damage are shown. This is handled by the "weapon" portal, which uses a Label script to synthesize its output.

The default output provided by the Skeleton files is perfectly reasonable, so there really isn't anything we need to do here.

Combat Actions To the left of the basic character information, a number of buttons will appear during combat. These buttons allow you to control whether a character has acted during combat. The Skeleton files provide support for three different actions, since that's common for many of the major game systems. The actions supported are "act", "delay", and "wait".

Every game system makes use of the "act" actions. This applies whenever the character takes his turn during combat. The other two actions may or may not apply to a given game system, and they may need to be adapted in some way. The d20 System has the concepts of "readying" an action and "holding" an action. The World of Darkness system has distinct concepts of delaying an action until later in the combat turn and delaying an action until the following turn.

In the case of Savage Worlds, though, there is only the "hold" option. Since initiative begins anew every combat round, there is no need for anything more than a simple "hold" action. This "hold" action can persist through the following rounds if the player chooses, so we need to track that state. We can easily do so by setting the existing "abandon" flag for any character that uses this action.

The "hold" option is probably best represented visually by the button that looks like "pause" on a VCR. Consequently, we'll use the "wait" portal within the "tacPick" template for the "hold" action. The "delay" portal can either be deleted or simply hidden. It's a little more work to delete it, but we'll go ahead and do that. The deletion requires that we remove all references to the portal within the Position script. This entails also positioning the lone "wait" portal properly, with the "next" portal relative to that. When in combat, we must adjust the left edge by the "next" portal instead of the "wait" portal, too

After revising the visuals, we also need to tweak the behaviors slightly. The Trigger script for the "wait" portal does exactly what we need, which is to set the field as an indicator and defer action for the character. However, the Trigger script for the "actnow" portal needs to be revised. Since there is only the one "hold" action, clicking this portal always acts after holding, so we must always reset the "abandon" field to zero. The resulting Trigger script should look like below.

```
herofield[acAbandon].value = 0
perform hero.combatact
```

The final thing we need to do is modify the MouseInfo scripts and/or "tiptext" attributes for the various portals. The behaviors are different, so we should explain the proper behaviors for Savage Worlds.

INITIATIVE HANDLING

During combat, an increment appears on the far right where you can manipulate the initiative for each character. For most game systems, initiative is a relatively quick and painless process. However, Savage Worlds uses a very different approach to initiative - a deck of cards. The Kit can support this non-standard mechanism, but it's not nearly as easy as a more traditional initiative. Since integrating the proper initiative for Savage Worlds will entail quite a few steps, we'll cover the initiative in a completely separate entry in this walk-through.

DAMAGE AND OTHER ACTOR MANIPULATIONS

To the lower right of the basic character information, a button appears with a skull on it. Clicking this button brings up a form where various in-play adjustments can be made directly to the character. This includes damage, resource tracking, activation of abilities, and even the tracking of combat notes.

The contents of this form are controlled in the file "form_manipulate.dat". However, it's unlikely you'll need to make any adjustments. The form utilizes the same layout that is defined for use on the "In-Play" tab. As such, any changes or refinements you make to the "In-Play" tab are automatically incorporated into this form. The only time you'll need to make changes is when you want to do something different for manipulation through the TacCon that doesn't appear on the "In-Play" tab. For our purposes, the form has everything we need on it, so there's nothing for us to do.

INITIATIVE (SAVAGE)

As mentioned in the previous section on the Tactical Console, initiative is usually pretty simple to implement. However, Savage Worlds uses a deck of playing cards for initiative. That requires a bit more work, so this topic works through the various steps involved in setting up such an initiative mechanism within HL.

ORIENTATION

Before we launch into the changes that need to be made, we'll start with a quick orientation to how initiative is managed within the Kit.

Every game system possesses some sort of mechanic for determining the order in which characters act during conflicts. Consequently, the Kit provides a built-in framework for handling

the process. The most common term used for the game mechanic is "initiative", so the Kit uses the same terminology.

The specifics of initiative range from the simple to the complex. To handle the full spectrum of games, the initiative framework within the Kit provide a fair amount of sophistication. Game systems that don't require all that sophistication can simply ignore what they don't need.

Internally, every character possesses an assortment of fields that are specific to initiative management. However, only two of these fields are exposed to you, as the author, for control. These fields track the primary initiative score for the character and a secondary tie-breaker score. As its name suggests, the tie-breaker is used to resolve situations where two characters possess the same primary initiative score. In many game systems, it's usual for ties to arise, so a mechanic is introduce to decide which character goes first.

The HL engine maintains the initiative for all characters. It automatically sorts characters into appropriate order based on the established rules for the game system. It also will automatically assign initiative values to characters, including the use of random die rolls when the game system calls for it. Provisions are made to allow users to either accept the assigned values or assign their own values. All of this is orchestrated via the Tactical Console, which is described in the User Manual for the product.

The data files are responsible for defining how the initiative score is determined for each character, as well as the tie-breaker. This is achieved via the Initiative script, which is defined within the file "definition.def". The script is invoked whenever an initiative score is needed for a character.

Two additional scripts can be defined for a game system. The "NewCombat" script is invoked for each character at the start of each new combat, while the "NewTurn" script is invoked for each character at the start of each combat turn. Both of these scripts provide the author with the opportunity to control state information for the characters. For example, many game systems have the concept of "holding" an action (although it may be called "delaying", "waiting", or something else). This state must be reset either at the start of a new combat or at the start of a new turn, and the two scripts make it easy to accomplish this.

You can dictate various characteristics of initiative for the game system. This is done via a number of attributes on the "behavior" element within the definition file. For example, you can control whether the game system generates new initiative values at the start of each turn or once at the start of the combat.

Lastly, there is a fourth script that can be defined for initiative. The vast majority of game systems use a simple numeric value to indicate the initiative score. Sometimes the score is generated randomly via a die roll, while other times it is calculated, but it is usually a number. In the few cases where the initiative score is not a number, you can define an "InitFinalize" script. This script behaves as a standard Finalize script that is applied to the initiative score for the character. This makes it possible to display the initiative to the user in whatever fashion is most appropriate for the game system.

Modeling a Deck of Cards

Savage Worlds used a deck of cards for initiative. This presents a number of wrinkles that we need to handle, the first of which is how to even model a deck of cards. There are 52 cards in a standard deck, plus the two jokers, for a total of 54 cards. This means that we can use a value in the range of 1 through 54 to represent the entire deck, with each value corresponding to a single card.

But we can't just use a random number generator, since all the cards in the deck must be unique and random numbers can be duplicated. We also need to effectively "shuffle" the deck to get all 54 values in a random order. Fortunately, the Kit provides a mechanism to generate a random set of values within a fixed range. This mechanism can be used for a variety of different purposes, but it's perfect for what we need.

Within the "state" script context, there are a number of target references that specifically pertain to managing random sets of values. The "setrandom" target reference allows us to create a new random set of values, giving it a name and the number of values to hold. We can use this to create a "deck" with 54 values in it, ranging from 0 to 53. The "setextract" target reference pulls a value (card) from the set (deck).

In the NewCombat script, we can create a newly shuffled deck of cards. In the Initiative script for each character, we can extract a card from the deck as our initiative value. The only consideration we have to worry about is that the NewCombat script is invoked for every character and we only want to shuffle a new deck once. This can be handled by keying on the "@isfirst" special symbol, which indicated when the first character is being processed.

Within the Initiative script, there are two special symbols that we need to assign. The "@initiative" special symbol represents that actual initiative value, so we extract a value from the set for that purpose. The "@tiebreaker" special symbol is used by HL to differentiate when two characters have the same initiative. Since a deck of cards is used, all values are guaranteed unique, so there is no tie-breaker in Savage Worlds. Consequently, we can set this symbol to zero.

Putting this together yields an initial two scripts as shown below.

```
<newcombat><![CDATA[
 ~if this is the first actor, shuffle a new deck for initiative
 if (@isfirst <> 0) then
   perform state.setrandom[deck,54]
   endif

 ~reset the abandon state in case it's still set from the previous
combat
  herofield[acAbandon].value = 0
 ]]></newcombat>

 <initiative><![CDATA[
   ~generate the primary initiative rating by drawing a card from
the deck
   @initiative = state.setextract[deck]

  ~we have no tiebreaker initiative rating since the cards are
always unique
   @tiebreaker = 0
   ]]></initiative>
```

After putting the above code into place, we can test our data files. Create a portfolio with a number of different characters in it, then show the Tactical Console. Every time we start a new combat, each character is randomly issued a new initiative value in the range of 0 to 53. There are never any duplicates due to the use of the set mechanism.

CONFIGURING THE INITIATIVE BEHAVIOR

When we did our testing above, there were a few things you probably noticed. First of all, a new initiative is only generated at the start of a new combat. Savage Worlds issues a new initiative at the start of each round, so we need to change that behavior. This is controlled via the "initperturn" attribute within the "behavior" element of the definition file. The default value is "no", so we need to specify a value of "yes".

While we're here, we should also impose an appropriate minimum and maximum value on the initiative range. The user is allowed to adjust the value freely within the TacCon, so negative values are possible and so are values larger than the number of cards in the deck. If we only have a deck of 54 cards to work with, we need to specify a minimum value of zero and maximum of 54. This is done via the "initminimum" and "initmaximum" attributes.

Applying these changes to the "behavior" element results in something that looks like below.

```
<behavior
  initperturn="yes"
  initminimum="0"
  initmaximum="53">
```

We also need to tell HL that the term used for each turn of combat within Savage Worlds is "round". We can accomplish this by specifying the "combatturnterm" attribute within the "structure" element of the definition file. By making this change, references to the "turn" within menus and the like will use the proper "round" term. The resulting "structure" element should look like the following.

```
<structure
  folder="savage"
  combatturnterm="round">
  </structure>
```

FORCING A RE-SHUFFLE

The next thing we need to deal with is when to re-shuffle the deck. We currently shuffle the deck at the start of the combat. However, we pull new cards from the deck every round, so we're bound to run out at some point. We also have to accommodate the rule that the deck is re-shuffled any round after a character draws a joker.

The fact that a joker is drawn can be tracked through a global state variable. Most everything with the HL is tied to a specific actor. However, in situations like this, you can set and retrieve state information that is global across all characters. Within the "state" script context, you can use the "value" target reference for this purpose. By specifying a unique id, you can save and retrieve a named value, which the following code demonstrates.

```
var temp as number
 state.value[joker] = 42
 temp = state.value[joker]
```

Let's put this technique to use. We only need to know whether a joker has been drawn of not, so we'll use a value of 1 to indicate a joker being drawn and a value of zero to indicate no joker. Whenever we re-shuffle the deck, we need to set the state value to zero to indicate no joker. Whenever we draw a card from the deck, we need to set the state value to one. We can accomplish this by revising the NewCombat and Initiative scripts to those shown below.

```
<newcombat><![CDATA[
  ~if this is the first actor, shuffle a new deck for initiative
  if (@isfirst <> 0) then
    perform state.setrandom[deck,54]
    state.value[joker] = 0
    endif

  ~reset the abandon state in case it's still set from the previous
combat
  herofield[acAbandon].value = 0
  ]]></newcombat>

<initiative><![CDATA[
  ~if this actor is holding his action, he does not get a new card
  if (herofield[acAbandon].value <> 0) then
    done
    endif

  ~generate the primary initiative rating by drawing a card from
the deck
  @initiative = state.setextract[deck]

  ~if we drew a joker, set the state accordingly
  if (@initiative >= 52) then
    state.value[joker] = 1
    endif

  ~we have no tiebreaker initiative rating since the cards are
always unique
  @tiebreaker = 0
  ]]></initiative>
```

NOTE! Since HL assumes that higher values go first for initiative, and since jokers are the best cards in Savage Worlds, we assume that the jokers are the highest card values (52 and 53).

Now we have to decide how we're going to handle things every round. The NewTurn script is invoked for every actor, but we only want to trigger a re-shuffle once. Fortunately, the NewTurn script has the same "@isfirst" special symbol that the NewCombat script possesses. This means we can check for whether to re-shuffle the deck within the NewTurn script, and we only do it for the first actor.

We need to trigger a re-shuffle whenever the "joker" state value is non-zero. However, we also need to re-shuffle in another situation. If don't have enough cards to pass out to all the actors, we need to re-shuffle the deck to ensure we do have enough cards. We can check this condition first by using the "setremain" target reference. If we don't have enough cards, we can simply set the "joker" state to one. When we then check the "joker" state, we'll then perform the shuffle if either condition occurred.

The NewTurn script shown below shows an implementation of this logic.

```
<newturn><![CDATA[
  ~if this is the very first actor for the turn, we've got some work
to do
  if (@isfirst <> 0) then

    ~if we don't have enough cards left for all actors, force a re-
shuffle
    if (state.actorcount > state.setremain[deck]) then
      state.value[joker] = 1
      endif
```

```
    ~if a joker has been pulled, re-shuffle the deck for initiative
    if (state.value[joker] <> 0) then
      perform state.setrandom[deck,54]
      state.value[joker] = 0
      endif
    endif
  ]]></newturn>
```

We can now reload the data files and put our changes to the test. If you want to make absolutely sure that the logic is performing the way you want it, you can insert a few "debug" statements into the scripts and watch the debug output while you start new combats and new turns.

Omitting Held Cards from Re-Shuffle

There is one important game mechanic that we have not addressed thus far. Characters who choose to "hold" their action may do so indefinitely. As long as they don't act, they hold onto the same card. This means that we need to ensure that any initiative cards dealt to actors on "hold" are not included in the next re-shuffle. If we don't, then the same card can be re-issued to a new actor.

The mechanism for managing random sets within the Kit does not allow us to omit cards from the deck. However, it does allow us to discard specific cards after a new deck is shuffled. The "setdiscard" target reference makes this possible.

After we re-shuffle the deck, we must go through all actors and identify the ones that are holding their action. The cards possessed by those actors must then be discarded from the newly shuffled deck. This will achieve the same set result as if we omitted the cards before shuffling. Integrating the code for this behavior into the NewTurn script results in the updated logic.

```
 <newturn><![CDATA[
   ~if this is the very first actor for the turn, we've got some work
to do
   if (@isfirst <> 0) then

    ~if we don't have enough cards left for all actors, force a re-
shuffle
     if (state.actorcount > state.setremain[deck]) then
       state.value[joker] = 1
       endif

    ~if a joker has been pulled, re-shuffle the deck for initiative
     var isshuffle as number
     if (state.value[joker] <> 0) then
       perform state.setrandom[deck,54]
       state.value[joker] = 0
       isshuffle = 1
       endif

    ~if we re-shuffled the deck, discard any cards possessed by
actors on hold
     if (isshuffle <> 0) then
       foreach actor in portfolio
         if (eachpick.herofield[acAbandon].value <> 0) then
           perform
state.setdiscard[deck,eachpick.herofield[tactInit].value]
           endif
         nexteach
       endif
     endif

   ~held actions persist from the previous round of combat, so
there's nothing to do
   ]]></newturn>
```

At this point, our logic is finally working the way it needs to.

Displaying Cards

The internal behaviors are in place, so we'll shift our focus to the presentation. On the Tactical Console, the initiative is managed via an incrementer. Within the incrementer, the cards are still being shown as a numeric value between 0 and 53. That's not very intuitive to players, and it's not going to work at all for GMs who want to actually deal out the cards in the game and then specify the initiative for each actor.

We need to convert the numeric value to a suitable card for display. The good news is that we can easily do so by defining an InitFinalize script. This script behaves like a standard Finalize script for fields, and it is applied to the field containing the initiative for each actor. If we implement this script correctly, the incrementer will show a traditional card instead of a numeric value.

We'll break the process up into two pieces. For each card, we need to determine the card symbol to be shown and the suit to be shown. We'll need to handle the jokers specially, since they don't have a suit.

We already determined that higher values are better, so that means the top two values belong to the jokers. Working downwards in value, the next four values are the aces, the next four the kings, and so on down to the deuces. Once we get to the single digit values, we can use math to quickly identify the specific character to be shown.

For the suit, we need to come up with something appropriate. We could easily us "S" for spades, "H" for hearts, etc. However, it would be even better if we could use the proper symbol for each suit. We can fire up the "Character Map" tool and take a look at the various fonts that are built into Windows. One of those fonts is the "Symbol" font, and within that font are characters for each of the different suits. Since the suits are progressive in nature, we can readily convert the card value to a value between 0 and 3. Once we do that, it's easy to convert it to the appropriate character code.

The last thing we do is combine the two pieces into the final text displayed. The whole process amounts to the InitFinalize script shown below, which we'll add just beneath the Initiative script.

```
 <initfinalize><![CDATA[
   var card as string
   var suit as string
   var temp as number

  ~determine the card symbol to be used
   if (@value >= 52) then
     card = "Jkr"
   elseif (@value >= 48) then
     card = "A"
   elseif (@value >= 44) then
     card = "K"
   elseif (@value >= 40) then
     card = "Q"    elseif (@value >= 36) then
     card = "J"    elseif (@value >= 32) then
     card = "10"
   else
     temp = (@value + 8) / 4
     card = chr(48 + round(temp,0,-1))
     endif

  ~determine the suit to be used
   if (@value >= 52) then
     suit = ""
   else
     temp = @value % 4
     suit = "{font Symbol}" & chr(167 + temp)
```

```
    endif

  ~assemble the final value
   @text = card & suit
  ]]></initfinalize>
```

Reload the data files and take a look at all the initiatives on the Tactical Console. Proper cards are now shown. If you use the up/down arrows on the incrementers to adjust the initiative, it will cycle through the cards of the deck in an intuitive manner.

Revising the Incrementer

The card symbols are appearing in the incrementer, but they don't look as good as they could. The font used and the size characteristics need to be tweaked. In addition, if the user clicks within the incrementer, the numeric value appear and the user can enter a custom value. This won't make any sense to the user, so we need to stop it. The solution to each of these is to come up with our own incrementer style that is tailored to the task.

We can open the file "styles_ui.aug" and locate that various existing incrementer styles. The "incrSimple" style is currently being used, so we'll clone that and then adapt it. We'll switch to the slightly smaller "fntincrsml" font and see how that looks. The card text looks good, so we'll stick with that. Now let's examine the horizontal spacing. We'll widen the text area a little bit, which means we must increase the full width and move the position of the "plus" arrow the same amount. Lastly, we need to make sure that the user can't directly edit the contents. Putting this all together yields the new style below.

```
 <style
   id="incrCard">
   <style_incrementer
    textcolor="f0f0f0"
    font="fntincrsml"
    editable="no"
    textleft="14" texttop="0" textwidth="28" textheight="20"
    fullwidth="54" fullheight="20"
    plusup="incplusup" plusdown="incplusdn" plusoff="incplusof"
    plusx="43" plusy="0"
    minusup="incminusup" minusdown="incminusdn"
minusoff="incminusof"
    minusx="0" minusy="0">
    </style_incrementer>
   </style>
```

Our style is defined, so we need to switch to using it. Modify the "totalinit" portal to use the new "incrCard" style and reload the files. That looks much better.

Refinements

Everything is working properly, but there are still a few refinements we need to make. For example, there are multiple places where the term "turn" is being used within text strings. These should all be changed to use the term "round" appropriately.

Next to the incrementer is an initiative adjustment. This is intended for use in game systems where characters have adjustments that apply to rolled initiative values. By having the adjustment visible, the GM can roll dice, apply the adjustment, enter the proper values for use. The problem is that this has absolutely zero pertinence to Savage Worlds.

We'll delete the "init" portal from the template. This requires that we revise the logic within the Position. We have to both eliminate all references to the portal and make sure that portals are re-positioned appropriately in the absence of the portal. The revised section of script code impacted by this removal is shown below.

```
 ~position the initiative incrementer right-aligned with the
activated abilities
  perform portal[totalinit].alignrel[rtor,active,-10]
  perform portal[totalinit].alignedge[bottom,-margin - 4]

 ~position the initiative label to the left of the incrementer
  perform portal[lblinit].alignrel[rtol,totalinit,-3]

 ~vertically align the initiative labels based on the incrementer
  perform portal[lblinit].centeron[vert,totalinit]

 ~all initiative details are only visible within combat
  show = state.iscombat  portal[totalinit].visible = show
  portal[lblinit].visible = show

 ~adjust our right edge leftward past the special abilities mouse-
over icon
  rightedge = portal[special].left - 4
```

We now have the initiative working properly for Savage Worlds and integrated smoothly into the Tactical Console.

Allies (Savage)

The Savage Worlds game system emphasizes the use of allies. As such, our data files should provide for the creation and management of allies associated with the characters.

Setting Up Ally Support

Allies are essentially independent characters that are children of the main characters. The Kit provides a framework for handling such characters very easily via the "minion" mechanism. Minions work very similarly to gizmos. They are attached to a character via a thing, and the minion is automatically added to the character whenever a pick based on that thing is added. Deleting the pick also deletes the minion. The parent character of a minion is referred to as the "master".

Just about any "thing" can have a minion attached. However, we want to be able to readily identify the picks that add our allies from other picks. We'll define a new component and component set to accomplish this. We'll assign the component a unique id of "Ally" and have it automatically define a component set with the same id, since we don't need to re-use any other special behaviors.

It would also be useful to let the user enter some arbitrary notes about each ally that can be used to readily identify them. For this, we'll include a field on the component where we can store those details. This results in a component set that looks like below, which we can add to the file "miscellaneous.str".

```
 <component
   id="Ally"
   name="Ally">

  <!-- Brief summary of ally for display in the list of allies -->
   <field
    id="alySummary"
    name="Summary"
    type="user"
    maxlength="100">
    </field>

  </component>
```

Defining the Minion

We can now define a thing based on our new component set. We can add the thing to the file "thing_miscellaneous.dat". The only item of note about this thing is that the the minion will be attached by it.

Associating a minion with a thing is accomplished via the "minion" child element. Each minion needs to be assigned a unique id, which makes it possible to identify different minions when multiple types of minions are added to a character. We'll only have one type of minion, but we need to assign a unique id anyway.

The other important facet of our minion is that we want it to inherit all of the settings associated with the master character. For example, if the master has only the "Futuristic" time period selected, then we want to assume that the minion has the same behaviors. This is achieved via the "isinherit" attribute within the "minion" element.

Putting this all together, we end up with a thing definition that looks like the following.

```
<thing
  id="mscAlly"
  name="Ally"
  compset="Ally">

 <minion
   id="ally"
   isinherit="yes">
   </minion>
 </thing>
```

Manipulating Allies

We now need to figure out how to let users add and manage allies. We could add allies to an existing tab, but none of them really seem appropriate. There also is a space consideration, as many of our tabs are already quite packed with information. Since we only have a rather small number of tabs, it would be quite reasonable to add another tab for tracking allies.

When adding our tab, we'll want something very simple. We'll have a single table on the tab where the user can add and access allies. The "Skills" tab is very similar, so we'll copy the file "tab_skills.dat" as "tab_allies.dat" and then adapt the file to our needs.

The first thing we need to do is revise the table portal at the top. All allies will be attached to the character via the same "mscAlly" thing that we defined above. Consequently, we need to utilize an "auto" table that automatically adds a new pick based on a specific thing instead of prompting the user to select a thing. This requires that we specify the thing id to be used. We also need to utilize a custom template for showing the contents of each ally. The resulting portal should look like the one shown below.

```
<portal
  id="alAllies"
  style="tblNormal">
  <table_auto
    component="Ally"
    showtemplate="alPick"
    autothing="mscAlly">
    <headertitle><![CDATA[
      @text = "Allies Associated with Character"
     ]]></headertitle>
    <additem><![CDATA[
      @text = "Add a New Ally to the Character"
     ]]></additem>
    </portal>
```

For the moment, we'll keep the template very simple. We'll start with just the name of the pick, plus the standard info and delete portals. We'll come back in a moment to refine the template and make it more useful. This yields a template like the one below.

```
<template
  id="alPick"
  name="Ally Pick"
  compset="Ally"
  marginhorz="3"
  marginvert="2">

  <portal
    id="name"
    style="lblNormal"
    showinvalid="yes">
    <label
      field="name">
      </label>
    </portal>

  <portal
    id="info"
    style="actInfo">
    <action
      action="info">
      </action>
    <mouseinfo/>
    </portal>

  <portal
    id="delete"
    style="actDelete"
    tiptext="Click to delete this item">
    <action
      action="delete">
      </action>
    </portal>

  <position><![CDATA[
    ~set up our height based on the tallest portal
    height = portal[info].height

   ~if this is a "sizing" calculation, we're done
    if (issizing <> 0) then
      done
      endif

   ~position our tallest portal at the top and center other portals
vertically
    portal[info].top = 0
    perform portal[name].centervert
    perform portal[delete].centervert

   ~position the delete and info portals on the far right
    perform portal[delete].alignedge[right,0]
    perform portal[info].alignrel[rtol,delete,-8]

   ~position the name on the left
    portal[name].left = 0
    ]]></position>
```

The next step is to revise the layout to show allies. All that entails is a switch to the new ids, which looks like below.

```
<layout
  id="allies">
  <portalref portal="alAllies" taborder="10"/>
```

```
  <position><![CDATA[
    ~position and size the table to span the full layout; it will only
use the
    ~vertical space that it actually needs
    perform portal[alAllies].autoplace
    ]]></position>

  </layout>
```

The final step is to modify the panel. We'll position the new tab between the "Personal" and "Journal" tabs, which means we need to assign it an order of 315. This yields the following panel.

```
<panel
  id="allies"
  name="Allies"
  marginhorz="5"
  marginvert="5"
  order="315">
  <layoutref layout="allies"/>
  <position><![CDATA[
   ]]></position>
  </panel>
```

We can now give things try. Reload the data files and you should see the new "Allies" tab. On the tab is a table, and clicking on the "add item" of the table automatically adds a new ally pick to the character. When this happens, you should also see a new character appear on the Dashboard. This is our new ally.

If you switch to ally via the Dashboard, you can see that the ally is a standard character. You can also verify that the various settings associated with the master character are properly inherited into the minion. At the top left of the minion, next to the name, a button should appear. This button allows you to quickly return to the master of the minion by clicking on it. Click the button and you should again be looking at the master character. Now delete the ally pick from the table, at which point our minion disappears. The basics of allies are now operational.

Revising the Template

We should now do something more useful that just show the name of our allies. One thing that would be extremely useful is to add a button that lets the user go directly to a particular ally. We can always rely on the Dashboard for this, but a button next to each ally would be much nicer.

We want to accomplish the exact same behavior as the Dashboard, and we should probably use the exact same button for consistency. So take a look at how the Dashboard accomplishes this. It uses a special action portal that handles all the mechanics automatically. We'll copy the portal into our template and re-use all that same logic.

When we defined the "Ally" component, we included a field where the user can specify details about the character. We should show an edit portal next to the name that allows the user to edit those details. We'll size the edit portal based on whatever space exists between the ally name and the "info" portal on the right.

This results in a revised template that looks like the one below.

```
<template
  id="alPick"
  name="Ally Pick"
  marginhorz="3"
  marginvert="2">

  <portal
    id="load"
    style="actLoad"
    tiptext="Click here to make this the active character.">
    <action
      action="minion">
      </action>
    </portal>

  <portal
    id="name"
    style="lblNormal"
    showinvalid="yes">
    <label
      field="name">
      </label>
    </portal>

  <portal
    id="summary"
    style="editNormal">
    <edit
      field="alySummary">
      </edit>
    </portal>

  <portal
    id="info"
    style="actInfo">
    <action
      action="info">
      </action>
    <mouseinfo/>
    </portal>

  <portal
    id="delete"
    style="actDelete"
    tiptext="Click to delete this item">
    <action
      action="delete">
      </action>
    </portal>

  <position><![CDATA[
    ~set up our height based on the tallest portal
    height = portal[info].height

    ~if this is a "sizing" calculation, we're done
    if (issizing <> 0) then
      done
      endif

    ~position our tallest portal at the top and center other portals
on it
    portal[info].top = 0
    perform portal[name].centeron[vert,info]
    perform portal[delete].centeron[vert,info]
    perform portal[load].centeron[vert,info]

    perform portal[summary].alignrel[btob,name,2]

    ~position the delete and info portals on the far right
    perform portal[delete].alignedge[right,0]
    perform portal[info].alignrel[rtol,delete,-8]

    ~position the load portal on the left, with the name and
summary adjacent
    portal[load].left = 0
    perform portal[name].alignrel[ltor,load,8]
    perform portal[summary].alignrel[ltor,name,10]
```

```
    portal[summary].width = portal[info].left - 10 -
portal[summary].left
    ]]></position>

  </template>
```

If we reload the files, we can use the button next to the name to go directly to a given ally, plus we can enter notes about the ally for easy access and viewing.

RECAP SUMMARY

Showing the name of each ally and a few summary notes is of limited use. What would be ideal is if we could actually show a detailed summary of each ally, much like the contents of a statblock. We could easily show this summary beneath the current information for each ally. However, we need to have the summary available.

We could generate the summary on-the-fly via a Label script. However, we may also want to show the recap else where. For example, within the mouse-info for the ally. In order to make sure we can have access to the recap from various places, we need to synthesize the results into a field. This field can be added to the "Actor" component and generated via an Eval script.

The contents of the recap summary should be as compact as possible. Consequently, we'll minimize spacing and punctuation to the minimum necessary. We'll also use abbreviations and short names wherever possible. The resulting field and script are demonstrated below.

```
 <field
   id="acRecap"
   name="Recap Summary"
   type="derived"
   maxlength="2000">
   </field>

 <eval index="5" phase="Render" priority="10000"><![CDATA[
   var txt as string
   var recap as string

   ~output any race
   txt = hero.firstchild["Race.?"].field[name].text
   if (empty(txt) = 0) then
     recap &= txt & ", "
     endif

   ~output the XP and rank
   var rankvalue as number
   var ranktext as string
   rankvalue = field[acRank].value
   call RankName
   recap &= ranktext & "  (" & #resmax[resXP] & " XP)"

   ~output attributes
   foreach pick in hero where "component.Attribute &
!Hide.Attribute"
     recap &= ", " & eachpick.field[trtAbbrev].text & " " &
eachpick.field[trtDisplay].text
     nexteach

   ~output derived traits
   foreach pick in hero where "component.Derived &  !Hide.Trait"
sortas explicit
     recap &= ", " & eachpick.field[trtAbbrev].text & " " &
eachpick.field[trtDisplay].text
     nexteach

   ~output special abilities
   foreach pick in hero where "component.Ability" sortas
     recap &= ", " & eachpick.field[shortname].text
     nexteach

   ~output arcane powers
   foreach pick in hero where "component.Power"
     recap &= ", " & eachpick.field[name].text
     nexteach

   ~output skills
   foreach pick in hero where "component.Skill &  !Hide.Skill"
     recap &= ", " & eachpick.field[trtAbbrev].text
     if (eachpick.tagis[User.NeedDomain] <> 0) then
       recap &= " (" & eachpick.field[domDomain].text & ")"
       endif
     recap &= " " & eachpick.field[trtDisplay].text
     nexteach

   ~save the final contents
   field[acRecap].text = recap
   ]]></eval>
```

Now that we've got the field being synthesized, we can put it to use. We need to add a new portal to the template for displaying the field. We'll allocate three lines of text to the recap for each ally, which should allow us to show a handful of allies at a time when the main window is at its smallest height. In the interest of keeping things as tight as possible, we'll also decrease the spacing between lines by a one pixel. This results in the following portal.

```
  <portal
    id="recap"
    style="lblSmlLeft">
    <label
      ismultiline="yes">
      <labeltext><![CDATA[
        ~output the recap field, but squeeze the line spacing a little
bit
        @text = "{leading -1}" & minion.herofield[acRecap].text
        ]]></labeltext>
      </label>
    </portal>
```

We need to factor the height of the new portal into our overall height for the template. Once that's done, we can then place the portal beneath the current line of portals, leaving a margin on each side to set it off better and clearly break up individual allies in the table. The pertinent changes and additions to the Position script are shown below.

```
  ~set up our height based on our full extent
   height = portal[summary].height + 5 + portal[recap].fontheight *
3

  ~position the recap portal beneath the top line and limit it to 3
lines
   perform portal[recap].alignrel[ttob,summary,3]
   portal[recap].lineheight = 3

  ~position and size the recap horizontally
   perform portal[recap].alignrel[ltol,name,0]
   portal[recap].width = portal[delete].left - 10 - portal[recap].left

  ~resize the contents of the recap portal if needed and ensure a
3-line height
   perform portal[recap].sizetofit[30]
   portal[recap].lineheight = 3
```

After reloading the data files, the recap text is quite helpful. Unfortunately, it's also still a bit too big and bold. It's competing with the primary information for each ally instead of being clearly supporting information. We need to switch to a different style that uses a smaller font and a less intense color. We could use a soft grey, but that's a little too subtle, so we'll choose a soft cyan instead. This yields the new style shown below, which can be swapped into use by the portal.

```
<style
  id="lblNotes">
  <style_label
    textcolor="99efed"
    font="fntnotes"
    alignment="left">
    </style_label>
  <resource
    id="fntnotes">
    <font
      face="Arial"
      size="34">
      </font>
    </resource>
  </style>
```

Refinements

Allies are basically working, but there are still a few things we should clean up. First of all, the mouse-info text shown for each ally is just the standard name and description text. We should really show the full recap information for the ally, since the three lines of space we've allocated may not be enough in some cases. This requires that we change the MouseInfo script from using the default behavior to more appropriate custom behavior.

We don't want to replace the standard behavior entirely, though. What we want is to append additional information at the end of the standard material. To accomplish this, we'll call the "MouseInfo" procedure to get the standard information and then append our own data at the end. This yields a new MouseInfo script that looks like the following.

```
<mouseinfo><![CDATA[
  var mouseinfo as string
  call MouseInfo
  @text = mouseinfo & "{br}{br}{b}Ally Summary:{/b}{br}" &
minion.herofield[acRecap].text
  ]]></mouseinfo>
```

Another issue with our implementation can be seen when we switch to an ally. The "Allies" tab is visible for our allies, which means that we can theoretically add allies to our allies, and those allies can have their own allies as well. While this is technically valid, it doesn't make sense within the context of a Savage Worlds game. So we need to hide the "Allies" tab for allies.

Hiding the "Allies" tab in general is easy, and we've done it before. First, we define a new "HideTab.allies" tag. Then we add a Live tag expression to the panel that verifies the tag doesn't exist on the character. But how do we get the tag onto the character properly?

Remember that minions work very much like gizmos. We can assign tags to a child entity within the definition by use of the "tag" element on the entity. We can do the same within our "minion" element. All we need to do is an additional line to our thing, which ends up looking like the following.

```
<thing
  id="mscAlly"
  name="Ally"
  compset="Ally">

  <minion
    id="ally"
    isinherit="yes">
    <tag group="HideTab" tag="allies"/>
    </minion>

  </thing>
```

If we reload the files now, we'll see the "Allies" tab appear normally for our main characters, but it disappears when we're manipulating an ally.

Character Sheet Additions (Savage)

Now that we've got allies working, we need to add them to the character sheet output. We should also provide details on vehicles within the character sheet. This wiki entry outlines how to accomplish this.

Printing Allies

Each ally can be printed out on a separate character sheet, just like a normal character. However, it would be much more convenient to print a compact representation of each ally along with the main character for which they are minions. We don't have room for allies on the first page of the character sheet, but we should be able to add them without any problem on the second page.

We'll add allies after everything else is output for the character. We can define a new table of allies and integrate it easily into the layout for the second sheet. In addition to printing all the basic details for each ally, we also need to include a separate space to track damage, power points, and ammo for each ally.

The Table Portal

Our table portal is a little bit different from the ones we normally define. We need our table to list all minions of the character. This can be done by operating on all picks in the "Ally" component, just like we did in the table on the tab. However, there is a better solution. We can instead specify that we want to list all minions via the "showpicks" attribute. This establishes the "actor" pick for the actual minion as the context for each item in the table, and that means that we can use the "hero" context readily within Label scripts in the template's portals.

For the sort set, we'll use the pre-defined "_ActorSeq_" sequence, which outputs the minions in an appropriate order for the character. Since our minions will not all be the same height, we need to specify that our height varies. Lastly, we don't want a header above the table. Instead, we'll draw our own header above each item in the table so that each minion looks like its own entry to the user. Consequently, we want to insert a gap between each item in the table, so we specify the "showgapy" attribute.

The net result of all of this is a table portal that looks like the following.

```
<portal
  id="oAlly"    style="outNormal">
  <output_table
    component="Ally"
    showtemplate="oAllyPick"
```

```
      showpicks="minion"
      showsortset="_ActorSeq_"
      varyheight="yes"
      showgapy="25">
      </output_table>
    </portal>
```

Layout Integration

Integrating our new portal into the layout is trivial. We simply add a new "portalref" for the portal and then add a suitable "autoplace" of the portal within the Position script. The revised layout looks like the following.

```
<layout
  id="oStandard2">
  <portalref portal="oPower"/>
  <portalref portal="oArmor"/>
  <portalref portal="oWeapon"/>
  <portalref portal="oGear"/>
  <portalref portal="oAlly"/>
  <position><![CDATA[
    ~position the various tables in the desired sequence
    perform portal[oPower].autoplace
    perform portal[oArmor].autoplace
    perform portal[oWeapon].autoplace
    perform portal[oGear].autoplace
    perform portal[oAlly].autoplace

   ~the height of the layout is the bottommost extent of the
elements within
    height = autotop
    ]]></position>
  </layout>
```

The Ally Template

The template itself will consist of four pieces that will each be handled by a separate portal. At the top of each ally, we'll display a title that looks like a standard title above a table. Beneath the title, we'll output all of the details for the character, including traits, abilities, and gear. At the bottom, we'll present a convenient place for tracking health and other status for the character. Between the details and tracking sections, we'll insert a solid line as a separator.

The overall template will look like the one shown below. This template omits the contents of the Label scripts that will be employed for three of the portals. Those contents will be discussed in the following sections.

```
<template
  id="oAllyPick"
  name="Output Allies Table"
  compset="Actor">

  <portal
    id="name"
    style="outTitle">
    <output_label>
      <labeltext><![CDATA[ ~insert script here ]]></labeltext>
      </output_label>
    </portal>

  <portal
    id="details"
    style="outAlly">
    <output_label>
      <labeltext><![CDATA[ ~insert script here ]]></labeltext>
      </output_label>
    </portal>

    id="line"
    style="oSeparator">
    <output_separator
      isvertical="no"/>
    </portal>

  <portal
    id="tracking"
    style="outAlly">
    <output_label>
      <labeltext><![CDATA[ ~insert script here ]]></labeltext>
      </output_label>
    </portal>

  <position><![CDATA[
    ~our name spans the entire width and is limited to a single
line in height
    portal[name].width = width
    portal[name].lineheight = 1

   ~position the details beneath the name, spanning the full
width
    perform portal[details].alignrel[ttob,name,10]
    portal[details].width = width
    perform portal[details].autoheight

   ~position the line beneath the details, spanning the full width;
we set the
    ~height to double the border size so that no contents are
visible and we
    ~end up with simply a solid horizontal line
    perform portal[line].alignrel[ttob,details,8]
    portal[line].width = width
    portal[line].height = portal[line].bordersize * 2

   ~position the tracking beneath the line, spanning the full
width
    perform portal[tracking].alignrel[ttob,line,5]
    portal[tracking].width = width
    perform portal[tracking].autoheight

   ~our final height is the bottom of the template contents
    height = portal[tracking].bottom
    ]]></position>   </template>
```

Showing the Title

The "title" portal must show a suitable name for ally. Each minion has a name, and that name is initialized to the name of the anchor pick for the minion. Consequently, if the user has not changed the actor's name, the two will match. If the name has been changed, we want to designate the nature of the minion in parentheses after that name. If the name has not been changed, we need to omit the nature, else we'll end up duplicating the default name twice (e.g. "Ally (Ally)").

We'll start with the name of the actor. If the name differs from the default name (accessed via the "miniontext" target reference), then we'll append the nature. This results in the Label script shown below.

```
@text = hero.actorname
if (empty(hero.miniontext) = 0) then
  if (compare(hero.miniontext,@text) <> 0) then
    @text &= " (" & hero.miniontext & ")"
    endif
  endif
```

Character Details

The core details for the character are rather extensive. We need to show the basics like the name and rank, as well as the attributes,

derived traits, abilities, and skills. To ensure that the ally is highly usable during play, we also need to include arcane powers, weapon details, armor, and gear.

The output format is very simple and compact. Each category of information is identified by an indicator in bold. The actual information follows in plain text. Each new category starts on a new line so that everything is cleanly organized and easy for the user to access whatever information he wants.

At the very top of the details section, we include any notes that the user assigned to the ally. Since our script starts with the "actor" pick of the ally as its initial context, we need to access the anchor pick that attaches the minion. Once we have the anchor pick, we can then access the appropriate field to include in our output.

The complete script to synthesize all the character details is shown below.

```
 var txt as string
  var ismore as number

 ~output any notes for the ally
  if (anchor.field[alySummary].isempty = 0) then
    @text &= "{b}Notes:{/b} " & anchor.field[alySummary].text &
"{br}"
   endif

~output any race
 txt = hero.firstchild["Race.?"].field[name].text
 if (empty(txt) <> 0) then
   txt = "-none-"
  endif
 @text &= "{b}Race:{/b} " & txt

~output the rank and XP var rankvalue as number
 var ranktext as string
 rankvalue = herofield[acRank].value
 call RankName
 @text &= "; {b}" & ranktext & "{/b} ("
 @text &= hero.child[resXP].field[resMax].text & " XP)"
 @text &= "{br}"

~use a hanging indent from here on out
 @text &= "{indent -100}"

~output attributes
 @text &= "{b}Attributes:{/b} "
 ismore = 0
 foreach pick in hero where "component.Attribute &
!Hide.Attribute"
   if (ismore <> 0) then
     @text &= ", "
    endif
   @text &= eachpick.field[trtAbbrev].text & " " &
eachpick.field[trtDisplay].text
   ismore = 1
   nexteach
 @text &= "{br}"

~output derived traits
 @text &= "{b}Traits:{/b} "
 ismore = 0
 foreach pick in hero where "component.Derived &  !Hide.Trait"
sortas explicit
   if (ismore <> 0) then
     @text &= ", "
    endif
   @text &= eachpick.field[trtAbbrev].text & " " &
eachpick.field[trtDisplay].text
   ismore = 1
   nexteach @text &= "{br}"
```

```
 if (hero.haschild["component.Ability"] <> 0) then
   @text &= "{b}Abilities:{/b} "
   ismore = 0
   foreach pick in hero where "component.Ability" sortas
SpecialTab
     if (ismore <> 0) then
       @text &= ", "
      endif
     @text &= eachpick.field[shortname].text
     ismore = 1
     nexteach
   @text &= "{br}"
   endif

~output arcane powers (if any)
if (hero.haschild["component.Power"] <> 0) then
   @text &= "{b}Arcane Powers (" & #trkmax[trkPower] & "):{/b} "
   ismore = 0
   foreach pick in hero where "component.Power"
     if (ismore <> 0) then
       @text &= ", "
      endif
     @text &= eachpick.field[name].text
     ismore = 1
     nexteach
   @text &= "{br}"
   endif

~output skills
 @text &= "{b}Skills:{/b} "
 ismore = 0
 foreach pick in hero where "component.Skill &  !Hide.Skill"
   if (ismore <> 0) then
     @text &= ", "
     endif
   @text &= eachpick.field[trtAbbrev].text
   if (eachpick.tagis[User.NeedDomain] <> 0) then
      @text &= " (" & eachpick.field[domDomain].text & ")"
      endif
   @text &= " " & eachpick.field[trtDisplay].text
    ismore = 1
    nexteach
  @text &= "{br}"

 ~output weapons (if any)
  if (hero.haschild["component.WeaponBase"] <> 0) then
    @text &= "{b}Weapons:{/b} "
    ismore = 0
    foreach pick in hero where "component.WeaponBase" sortas
Armory
      if (ismore <> 0) then
        @text &= ", "
       endif
      @text &= eachpick.field[name].text
      @text &= " " & eachpick.field[wpNetAtk].text
      @text &= " (" & eachpick.field[wpShowDmg].text
      if (eachpick.field[wpPiercing].value <> 0) then
        @text &= ", AP" & eachpick.field[wpPiercing].text
        endif
      if (eachpick.tagis[component.WeapRange] <> 0) then
        @text &= ", " & eachpick.field[wpRange].text
        endif
      @text &= ")"
      ismore = 1
      nexteach
    @text &= "{br}"
    endif

 ~output armor and gear together (if any)
  if (hero.haschild["component.Defense | component.Equipment"]
<> 0) then

    ~output armor
     ismore = 0
```

```
    foreach pick in hero where "component.Defense" sortas
Armory
      if (ismore <> 0) then
        @text &= ", "
        endif
      @text &= eachpick.field[name].text
      @text &= " (" & signed(eachpick.field[defDefense].text)
      if (eachpick.tagis[component.Shield] <> 0) then
        if (eachpick.field[defParry].value <> 0) then
          @text &= ", Parry" & signed(eachpick.field[defParry].text)
          endif
        endif
      @text &= ")"
      ismore = 1
      nexteach

    ~output gear
     foreach pick in hero where "component.Equipment"
      if (ismore <> 0) then
        @text &= ", "
        endif
      @text &= eachpick.field[grStkName].text
      ismore = 1
      nexteach
     @text &= "{br}"
     endif
```

Status Tracking

At the very bottom of each ally, we're going to allows users to easily track the status of the ally during play. We'll include tracking for the ally's health, power points (if applicable), and ammunition (if applicable). The health tracking will depend on whether the character is a wildcard or not. For power points, we'll simply show a series of boxes, up to the total number of power points possessed by the character. The ammo will use the simplified mechanism outlined in the rulebook, which presents four levels.

The resulting Label script for synthesizing all the tracking information is shown below.

```
  var box as string
  var gap as string

 ~output mechanism for tracking health
  gap = "{horz 50}"
  @text &= "{b}Wounds:" & gap
  if (herofield[acIsWild].value <> 0) then
    @text &= "-1" & gap & "-2" & gap & "-3" & gap
    endif
  @text &= "INC"
  @text &= "{horz 100}Fatigue:" & gap
  @text &= "-1" & gap & "-2" & gap & "INC"
  @text &= "{/b}{br}"

 ~output boxes for tracking power points (if applicable)
  if (hero.haschild["component.Power"] <> 0) then
    @text &= "{vert 5}"
    @text &= "{b}Power:{/b}{horz 40}"
    var i as number
    var j as number
    var limit as number
    limit = #trkmax[trkPower] / 5
    @text &= "{font wingdings}"
    for i = 1 to limit
      for j = 1 to 4
        @text &= chr(168)
        next
      @text &= "{size 44}" & chr(168) & "{size 36}"
      next    @text &= "{revert}{br}"
    endif
```

```
 ~output boxes for tracking the ammo supply (if any ranged
  if (hero.haschild["component.WeapRange"] <> 0) then
    @text &= "{vert 8}"
    box = "{font Wingdings}{size 40}" & chr(168) & "{revert}"
    gap = "{horz 65}"
    @text &= "{b}Ammo:{/b}" & gap & "Full " & box & gap & "High
" & box & gap & "Low " & box & gap   & "Out " & box
    endif
```

Printing Vehicles

We currently just identify the existence of vehicles within the gear list. This is adequate when the vehicle is used as nothing more than a form of transportation. However, Savage Worlds includes full rules for vehicles in combat, and many of the vehicles are specifically intended for use in combat. Consequently, there will often be times when a player wants to see full details for his vehicles.

We can incorporate vehicles into the character sheet in the exact same way as we handle allies. We'll use the exact same approach, showing the vehicle name as a title at the top, with the vehicle details and damage tracking beneath. Since we're using the same approach, we can copy what we have for allies and adapt it for use with vehicles.

The Table Portal

We'll start with the table portal. For vehicles, our table portal will be operating on picks within the character instead of separate minions. This means that we can use the default behaviors for both what to show in the table and how to sequence them. We'll still want our items to vary in height, and we'll provide our own title at the top of each, so we'll also need the gap between items. This results in the table portal below.

```
<portal
  id="oVehicle"
  style="outNormal">
  <output_table
    component="Vehicle"
    showtemplate="oVehPick"
    varyheight="yes"
    showgapy="25">
    </output_table>
  </portal>
```

Layout Integration

Integrating this new table portal into the layout will work the exact same way as allies. We need to add the "portalref" element and include an "autoplace" statement to position the new table. We can add our vehicles either before or after allies, so we'll choose to add them afterwards. This results in the following revised layout.

```
<layout
  id="oStandard2">
  <portalref portal="oPower"/>
  <portalref portal="oArmor"/>
  <portalref portal="oWeapon"/>
  <portalref portal="oGear"/>
  <portalref portal="oAlly"/>
  <portalref portal="oVehicle"/>
  <position><![CDATA[
    ~position the various tables in the desired sequence
    perform portal[oPower].autoplace
    perform portal[oArmor].autoplace
    perform portal[oWeapon].autoplace
    perform portal[oGear].autoplace
```

```
    perform portal[oAlly].autoplace
    perform portal[oVehicle].autoplace

   ~the height of the layout is the bottommost extent of the
elements within
    height = autotop
   ]]></position>
  </layout>
```

THE TEMPLATE

Our template will behave the exact same was as for allies. We'll have the same four sections that serve the same roles. As such, our Position script requires zero changes. The only things we need to change are the unique id, name, and the contents of three of the portals.

The first portal outputs the name of the vehicle. Since we can assume that the names of vehicles will make each entry readily identifiable as a vehicle, we don't have to do anything special with the name. So we'll change the portal to a simple field-based portal, as shown below.

```
<portal
  id="name"
  style="outTitle">
  <output_label
   field="name">
   </output_label>
  </portal>
```

The vehicle details are completely different from allies. However, we're going to use the identical approach in synthesizing the output. We can replace the Label script for the portal with the appropriate information for vehicles. If a vehicle has weapons, we need to include full details for each weapon as well. This results in the following revised portal for vehicles.

```
<portal
  id="details"
  style="outAlly">
  <output_label>
   <labeltext><![CDATA[
    ~output the crew size
    @text &= "{b}Crew:{/b} " & field[vhCrew].text & "{br}"

    ~output the acceleration, speed, and climb for aircraft
    @text &= "{b}Acceleration{/b}: " & field[vhAccel].text
    @text &= "{horz 75}{b}Top Speed:{/b} " &
field[vhTopSpeed].text
    if (tagis[VehType.Aircraft] <> 0) then
     @text &= "{horz 75}{b}Climb:{/b} " & field[vhClimb].text
     endif
    @text &= "{br}"

    ~output the toughness and armor
    @text &= "{b}Toughness{/b}: " & field[vhTough].text
    @text &= "{horz 75}{b}Armor{/b}: " & field[vhArmor].text &
"{br}"

    ~if there is no child entity/gizmo, then there's nothing more
to do
    if (isentity = 0) then
     done
     endif

    ~use a hanging indent from here on out
    @text &= "{indent -150}"

    ~there is a child entity/gizmo, so output the load-out
    foreach pick in gizmo
     @text &= "{horz 10}" & chr(149) & " " &
eachpick.field[name].text & ": "
     @text &= eachpick.field[wpShowDmg].text & ", " &
eachpick.field[wpRange].text
     if (eachpick.field[wpNotes].isempty = 0) then
      @text &= ", " & eachpick.field[wpNotes].text
      endif
     @text &= "{br}"
     nexteach
    ]]></labeltext>
   </output_label>
  </portal>
```

The portal for the separator line requires no changes, so our final portal to deal with is for status tracking during play. Vehicles only need a damage track, so that's what we'll include. Tracking ammunition for weapons with hundreds or thousands of rounds is not practical on the character sheet, so we won't even try. This yields the revised portal below.

```
<portal
  id="tracking"
  style="outAlly">
  <output_label>
   <labeltext><![CDATA[
    var gap as string

    ~output mechanism for tracking damage
    gap = "{horz 50}"
    @text &= "{b}Damage:" & gap
    @text &= "-1" & gap & "-2" & gap & "-3" & gap & "Wrecked"
    @text &= "{/b}"
    ]]></labeltext>
   </output_label>
  </portal>
```

Vehicle output within character sheets is now fully operational.

USER-CONFIGURABLE OPTIONS (SAVAGE)

In most game systems, there are a number of optional game mechanics that may be employed. These range from methods for character creation to rules for combat resolution. There are also situations where you'll want to provide users with the ability to customize the contents of character sheets. All of these have one thing in common: the ability for users to enable options that influence how the data files behave.

All user-configurable options are presented to the user within the "Configure Hero" form. These options can be organized into logical groups to make access easier for users. The sections below introduce various ways to utilize user-configurable options within the Savage Worlds data files.

HOW SOURCES WORK

The term "source" is used because most user-configurable settings will be tied to specific rule supplements for a particular game system. Each of those supplements is a different source of game material, with its own unique characteristics. To keep things organized for the user, the various options are organized into a hierarchy. To keep things simple for the author, the same "source" elements are used for each node in the hierarchy, with different attributes being assigned to dictate the unique behaviors of each node.

Source elements can be designated as children of other sources via the "parent" attribute. This is how you construct the overall

hierarchy of sources. If the source "chlid" needs to be shown beneath the source "grouping" in the hierarchy, you would specify the "parent" attribute on the "child" source as "grouping". This approach makes it possible to add new sources to an existing grouping without having to modify the parent. When other authors want to extend your data files by adding support for other source books (or their own game world), this technique allows them to do so independently of your files.

Sources that are intended for user-selection must be designated as such via the "selectable" attribute. Only sources that possess no children are allowed to be selectable. If you want to force the selection of a source, then you can make it non-selectable and mark is as automatically selected.

That brings us to one last configurable facet of sources that we'll cover here. Sources can be marked as selected by default via use of the "default" attribute. By default, sources are not selected until the user selects them. However, you can reverse that behavior so that the source is selected unless the user deselects it. This is useful in situations where you believe most users will normally want to utilize an option. They can turn it off if they want, but the option is enabled if they don't specifically do so.

The question that remains is how sources can be utilized within the data files. Every source has a tag automatically defined by the Kit. These tags always belong to the group with the unique id "source" and possess a tag id matching the id of the source. Therefore, if you define the source "pickme", the tag id "source.pickme" will be automatically defined.

Whenever the user enables the source, the tag is automatically assigned to the hero, and the tag is not assigned then the source is deselected. This makes it possible to easily check the hero for the various source tags to determine if an option has been selected or not. You can reference source tags anywhere within your data files, such as testing them within scripts. You can even use them within ContainerReq and/or Live tag expressions, allowing you to customize the interface or control whether things exist within the data files.

DARK CAMPAIGN SETTINGS

The original Savage Worlds rulebook provides some suggestions regarding different campaign settings. For example, there are two "Dark Settings" presented. In a "Flawed Heroes" setting, heroes are allowed to take an additional Minor Hindrance at character creation, while a "Tragic Heroes" setting allow heroes to take an extra Major Hindrance.

Our current data files will report any attempt to do take extra hindrances as a validation error. What we need is the ability for users to toggle these options on if they wish, with our validation rule making the appropriate adjustments. We can model these options easily via the Sources mechanism.

The first thing we need to do is setup a source that will serve as a logical grouping for the user. Each of these options reflects the nature of the campaign setting being played, so that's what we call the grouping. Since this is intended as a grouping, we need to make it non-selectable. This results in the new source shown below, which we can add to the file "control.1st".

```
<source
  id="CampType"
  name="Campaign Settings"
  description="Options associated with the style of campaign
setting being played">
  </source>
```

The next thing we need to do is setup our individual options as sources. We need one option for flawed heroes and another for tragic heroes. Both sources need to reference the "Settings" source we defined above as their parent. This yields the following two new sources.

```
<source
  id="Flawed"
  name="Flawed Heroes"
  parent="CampType"
  description="Heroes may select one additional Minor
Hindrance, earning one extra Hindrance point">
  </source>

<source
  id="Tragic"
  name="Tragic Heroes"
  parent="CampType"
  description="Heroes may select one additional Major
Hindrance, earning two extra Hindrance points">
  </source>
```

Our sources are defined and should now appear for the user to select. However, we still need to do something useful with them. The validation test that checks on the limit of hindrances is controlled via the "valHinders" thing, which is defined in the file "thing_validation.dat". We'll need to modify the validation logic to incorporate the handling of these two sources.

Within the Eval Rule script, we assume that there is a maximum of one major and two minor hindrances allowed. We need to change this so that we start with those numbers by default and add in an extra one if the appropriate source tag is defined. We also need to synthesize the validation message via the script so that we can show the proper maximum limits based on whether the options are selected by the user. This yields a revised Eval Rule script that looks like the one below.

```
<evalrule index="1" phase="Validate" priority="8000"
    message="Maximum limits exceeded" summary="Limit
exceeded"><![CDATA[
  ~iterate through all hindrances and tally up the number of
majors and minors
  var major as number
  var minor as number
  foreach pick in hero where "component.Hindrance"
    if (eachpick.field[hinMajor].value = 0) then
      minor += 1
    else
      major += 1
    endif
  nexteach

  ~determine our maximum number of major and minor
hindrances
  var max_major as number
  var max_minor as number
  max_major = 1 + hero.tagis[source.Tragic]
  max_minor = 2 + hero.tagis[source.Flawed]

  ~if we have no more than our maximum major and minor
hindrances, we're good
  if (major <= max_major) then
    if (minor <= max_minor) then
      @valid = 1
```

```
      done
      endif
    endif

  ~synthesize our validation message appropriately
   @message="Maximum of " & max_major & " major and " &
max_minor & " minor hindrances allowed"

  ~mark associated tabs as invalid
   container.panelvalid[edges] = 0

  ~assign a tag to the hero to indicate the invalid state

  ~Note: This is used to color highlight the title above hindrances
on the tab
.   perform hero.assign[Hero.BadHinders]
   ]]></evalrule>
```

Reload the data files and give our changes a try. Add an extra major hindrance to the character and the validation error should appear. Then selects the "Tragic Heroes" option and the validation error should disappear. Everything is now working as it should.

Heroic Campaign Settings

The original rulebook also a few other campaign setting changes that apply in a heroic or cinematic setting. Two of these adjustments impact character creation and should be incorporated into our data files. The first is for a "Heroic" setting, wherein characters can ignore Rank requirements for edges taken during character creation. The second is an "Epic Heroism" setting, in which all Rank requirements can be ignored.

The first step in implementing these options is to set them up as sources. We'll do this just like we did in the previous example, adding one source for each of the two settings. Both sources must reference the "CampType" source we defined previously, resulting in the two new sources shown below.

```
 <source
   id="Heroism"
   name="Heroism"
   parent="CampType"
   description="Heroes ignore the Rank requirements for Edges
during character creation">
   </source>

 <source
   id="EpicHero"
   name="Epic Heroism"
   parent="CampType"
   description="Heroes ignore Rank requirements for Edges at
all times">
   </source>
```

With our sources defined, we can now shift our focus to properly handling their selection. The pre-requisite handling for the rank is handled within a PreReq test that is defined on the "MinRank" component. We'll find this component in the file "components.core", where we can modify it to support these new sources.

Within the Validate script of the pre-requisite test, the character's rank is tested against the requirements for the item. If the rank test fails, we then need to perform additional tests. This component is used within multiple different component sets, but our sources only apply to Edges. As such, all of our extra tests must only be performed for edges.

If the "Epic Heroism" source is selected, then an edge is always considered to be valid, so we can handle that quickly and easily. However, the "Heroism" source is only applied to edges selected during character creation, so we must determine whether our item satisfies that requirement. Each pick tracks whether it was added during character creation, with the state being accessible via the "creation" target reference. Things don't possess such a state, though, so we instead need to check the global state to determine whether we're in "creation" mode.

Putting all of this logic together results in a revised Validate script that looks like the one below.

```
 <validate><![CDATA[
   var rankvalue as number
   var ranktext as string

  ~get the minimum rank required for the thing to be valid
   rankvalue = altthing.field[rnkMinRank].value

  ~if the minimum rank is satisfied, we're good to go
   if (herofield[acRank].value >= rankvalue) then
     @valid = 1
     done
     endif

  ~if this is an edge, check for special configuration options
   if (altthing.tagis[component.Edge] <> 0) then

    ~if this is an epic heroism setting, we're valid
     if (hero.tagis[source.EpicHero] <> 0) then
       @valid = 1
       done
       endif

    ~determine whether this edge is added during creation
     var iscreate as number
     if (@ispick = 0) then
       iscreate = state.iscreate
     else
       iscreate = altpick.creation
       endif

    ~if this is an heroic setting and this is an edge at creation,
we're valid
     if (hero.tagis[source.Heroism] + iscreate >= 2) then
       @valid = 1
       done
       endif
     endif

  ~mark the panel as invalid
   altthing.linkvalid = 0

  ~synthesize an appropriate validation error message
   call RankName
   @message = ranktext & " rank required."
   ]]></validate>
```

Original Rulebook Options

The previous two sections outlined options that were described in the original rulebook for the game. As such, if a user sees them in the list and doesn't know about the original rulebook, he'll wonder where these options come from. So it's best if we add a new source that identifies itself as containing options from the original rulebook. We can then change the sources above to be children of this new source. The new source should look something like the one shown below.

```
<source
  id="Original"
  name="Original Rulebook Options"
  selectable="no"
  reportable="no"
  description="Assorted options that were outlined in the
original core rulebook">
  </source>
```

Once the new source is added, we can modify the "CampType" source to specify our new source as its parent, as shown below.

```
<source
  id="CampType"
  name="Campaign Settings"
  parent="Original"
  selectable="no"
  description="Options associated with the style of campaign
setting being played">
  </source>
```

FILTERING EQUIPMENT

The various lists of equipment (gear, weapons, etc.) are quite long. They also cover a wide range of time periods. If a user is creating a character for a game set in the 1700s, it's probably going to be annoying to wade through all the options for computers, surveillance gear, etc. What we need is a way to break the equipment up into a few smaller groupings, and sources provide exactly the solution we need.

The rulebook defines four general classifications for time period. These are: medieval, black powder, modern, and futuristic. We'll define new sources corresponding to each of these periods, plus an additional source to use as their parent. This results in the following new definitions.

```
<source
  id="TimePeriod"
  name="Time Period"
  selectable="no"
  description="Time periods and Savage Settings that govern
the availability of equipment">
  </source>

<source
  id="TimeMedi"
  name="Medieval"
  parent="TimePeriod"
  default="yes"
  description="Enables the availability of equipment from the
Medieval time period">
  </source>

<source
  id="TimePowder"
  name="Black Powder"
  parent="TimePeriod"
  default="yes"
  description="Enables the availability of equipment from the
Black Powder time period">
  </source>

<source
  id="TimeModern"
  name="Modern"
  parent="TimePeriod"
  default="yes"
  description="Enables the availability of equipment from the
Modern time period">
  </source>
<source
  id="TimeFuture"
  name="Futuristic"
  parent="TimePeriod"
  default="yes"
  description="Enables the availability of equipment from the
Futuristic time period">
  </source>
```

Once all the sources are defined, we can now make each piece of equipment dependent on the appropriate sources. This is accomplished via the "usesource" element. For example, associating a piece of equipment with the "Modern" time period entails adding the following reference to the thing definition.

```
<usesource source="TimeModern"/>
```

Any equipment that should exist within all time periods does not require any "usesource" assignment. If a thing has no source restrictions, then it always appears everywhere. We only need to assign the source restriction to equipment that is only available in a subset of the time periods (e.g. computers only exist in modern and future periods). It is perfectly valid to assign multiple time periods to a particular piece of equipment. If any of the specified sources is selected by the user, the equipment will appear.

THINGS FROM ORIGINAL RULEBOOK

Throughout this walk-through, we've included a number of objects that were pulled from the original rulebook and don't exist in the Explorers Edition. For example, all of the races we added (other than human) and the various complex military vehicles. We need to add appropriate sources for these objects and then associate the corresponding "thing" elements to those sources.

We'll first define a source for races. This source will govern the visibility of the various races from the core rulebook, and it should look like the following.

```
<source
  id="OrigRace"
  name="Show Races"
  parent="Original"
  description="Enables selection of races from the original core
rulebook">
  </source>
```

We can now associate each of the races other than "Human" to this source. All we need to do is add the "usesource" reference shown below to each thing. Note that we do not need to assign the source dependency to the racial abilities. Since those abilities are only accessible via the race, they are implicitly omitted along with the race.

```
<usesource source="OrigRace"/>
```

We'll now do the same thing for vehicles. We'll define a source that controls their availability and then associate each vehicle with the source. The new source is shown below.

```
<source
  id="OrigVeh"
  name="Show Vehicles"
  parent="Original"
```

```
    description="Enables selection of vehicles from the original
core rulebook">
    </source>
```

We can now associate the source with each vehicle via the "usesource" element. Unfortunately, this doesn't seem to work. The problem is that the vehicles have multiple sources - both the new source and a time period. The way the source mechanism works is that only one of the defined sources must be selected for the source to be active. Some of the major game systems have rules that are duplicated in multiple supplements, so the rules should be enabled if any of those supplements is enabled.

In our case, we want to require that both sources be selected for the vehicle to be shown. To accomplish that, we need to leverage a ContainerReq test instead. This test simply checks for both sources within the tag expression, as shown below. We should delete the "usesource" elements when we add the ContainerReq test to avoid any confusion.

```
<containerreq phase="Initialize" priority="2000"><![CDATA[
  source.OrigVeh & source.TimeModern
  ]]></containerreq>
```

CHARACTER SHEET OUTPUT

One of our original objectives with the character sheet was to fit everything on a single page. All of the core character information achieves this now, unless the character is quite complex. The problem is that we've added the ally and vehicle output only to the second page of the character sheet, which forces the printing of a second page if either are present. It's quite possible that a user will want to print allies as entirely separate characters and will only use vehicles as a means of transportation, in which case a single-page character sheet is desired and neither will be wanted on the second page.

We accommodate this very easily through the use of a couple of new user-configurable options. We'll keep the printing of allies and vehicles separate, just in case a user wants to include one but not the other. So we'll need to define two new sources. There is already an "Output Options" source grouping with one source beneath it. All we need to do is add our new sources and designate the appropriate parent. Since we want users to see the extra output instead of assuming it doesn't exist, we also want to set the default state of both sources to selected. This yields the two new sources below.

```
 <source
   id="ShowAlly"
   name="Include Ally Summaries on Sheet"
   parent="Output"
   default="yes"
   description="If the hero has any allies, separate summaries
for each ally will be included on   secondary pages of the
character sheet">
   </source>

 <source
   id="ShowVeh"
   name="Include Vehicles Summaries on Sheet"
   parent="Output"
   default="yes"
   description="If the hero has any vehicles, separate summaries
for each vehicle will be included on secondary pages of the
character sheet">
   </source>
```

If we select these options for a new character, we'll notice something that needs to be tweaked. On the "Configure Hero" form, a summary of the selected options is shown. Each of our new options runs off the right edge of the available space. We can either shorten the overall name or we can make use of the "abbrev" attribute. This attribute allows us to specify a shorter name for a source that will be shown in places like the summary list. We can add the following two lines to the source definitions, which will give us much better results.

```
abbrev="Show Ally Summaries"

abbrev="Show Vehicle Summaries"
```

With the sources defined, we can now put them to use within the second page of the character sheet. Open the file "sheet_standard2.dat" and locate the layout. We need to add special handling for both the "oAlly" and "oVehicle" table portals. If the source tag is defined, we'll "autoplace" the portals normally. However, if the source tag is not defined, we'll set the portal to be non- visible. That's all there is to it. This results in a revised Position script that looks like the one below.

```
~position the various tables in the desired sequence
 perform portal[oPower].autoplace
 perform portal[oArmor].autoplace
 perform portal[oWeapon].autoplace
 perform portal[oGear].autoplace

~only include allies if the user has enabled them
 if (hero.tagis[source.ShowAlly] = 0) then
   portal[oAlly].visible = 0  else
   perform portal[oAlly].autoplace
   endif

~only include vehicles if the user has enabled them
 if (hero.tagis[source.ShowVeh] = 0) then
   portal[oVehicle].visible = 0
 else
   perform portal[oVehicle].autoplace
   endif
~the height of the layout is the bottommost extent of the
elements within
 height = autotop
```

SEQUENCING THE SOURCES

Our list of sources has become rather long. Since the list is sorted alphabetically, that puts the "Original Rulebook" source at the top of the list, but that group of options is the least likely to be used. Our list of time periods has a similar problem, in that the sequence is not chronological, which is confusing. We should dictate the order in which the various sources are shown. Fortunately, the Kit provides the ability to do this.

Each source has an optional "sortorder" attribute. If this attribute is specified, all of the sources within a given grouping (i.e. sharing the same parent) are first sorted on the assigned order, from lowest to highest. If the order value is the same, the sources are sorted alphabetically. Any source with no assigned order goes after all sources that do have an order.

We're only going to worry about the sources that need sequencing, since most of them can be left to their default behavior. For the various time period sources, we can simply assign the values one

through four, as appropriate. However, for the parent sources, we should number them with values that leave gaps. That way, we can easily insert new groupings between existing groupings if we want. We'll use the values listed below.

Time Period: 20
Output Options: 50
Original Rulebook: 100

Once we assign the appropriate "sortorder" attributes to the various sources, we can reload and see the list in a much more intuitive order.

SPECIALIZED EDGES (SAVAGE)

Early in the development of our data files, we determined that we would need to do some special handling for a few of the edges. For example, the "Professional" edge requires the user to select a trait that with a rating of d12. Similarly, the "Scholar" edge requires the user to select two skills with a rating of d8 or higher. We need to allow the user to select the appropriate choices via menus, plus we need to properly validate that everything has been done correctly.

BUILT-IN SUPPORT

Situations where the player must select a trait or two to associate with an ability are reasonably common across game systems. There are also many situations where a pick's state needs to be toggled or choices need to be selected from a dynamic list. Consequently, the Skeleton files provide a substantial amount of built-in support to handle these cases. This support takes the form of a component to handle the internal mechanics and a template for displaying picks with menu selections.

The "UserSelect" component provides support for all the mechanics. In general, this component can be used to handle the vast majority of the customizations that you'll need. The "UserSelect" component supports a variety of behaviors, including a toggle state that can be selected by the user (generally via a checkbox), a menu of choices that are driven dyanmically at run-time, and up to two different menus for selecting other things or picks within the data files. Between these four mechanisms, you should be able to use the "UserSelect" component almost all the time.

The Skeleton files also provide a "UserSelect" template within the file "visual.dat". This template is similar to the "SimpleItem" template, except that it is designed to orchestrate the display and handling of picks derived from the "UserSelect" component. Fields within the component are defined to tailor how the visuals behave, while the template interprets these fields to appropriately display information and allow the user to modify it.

You will almost certainly find situations where the "UserSelect" component will come in handy. When integrating the visual behaviors, you can either use the corresponding template or adapt aspects of it for use within your own templates. Either approach is perfectly reasonable and up to you as the author.

INTEGRATING USER-SELECTION

As mentioned above, the "UserSelect" component encapsulates most of the behaviors you'll need. We'll now review how those various behaviors work and how to tailor them appropriately. There are a total of three different behaviors.

The first behavior is a checkbox, which we've already used on a few occasions (e.g. equipping weapons and armor). The component provides two fields for managing the checkbox state. The "usrIsCheck" field tracks whether the checkbox is actually checked, and the "usrChkText" field dictates the text to be displayed with the checkbox. If this field is empty, then pick is assumed to not utilize the checkbox and it is not shown.

The second behavior is a thing-based menu, which populates the menu with a list of things or picks that can be chosen. The specific list of items shown is dictated by a tag expression that must be placed into the "usrCandid1" field. If this field is empty, it indicates that no thing-based menu is to be shown. You can control whether list to choose from consists of things, picks that have been added to the hero, or picks that have been added to the current container. This is accomplished by specifying one of the tags from the "ChooseSrc1" tag group, with no tag resulting the "hero" behavior.

Once the user selects an item, it is stored in the field "usrChosen1". If the user does not select an item, a validation error is automatically triggered. If you want to display a label next to the menu, you can specify the text to be shown in the "usrLabel1" field.

It is also possible to specify two separate thing-based menus at the same time. Two sets of the various fields are provided. In most cases, only one menu will be needed, but there will be times that two are required. When that occurs, you can use the second set of fields. They work exactly as the first set, although the first set of fields must be specified before you use the second set. This means that the field "usrCandid1" must be defined if you want to also use "usrCandid2".

The third behavior is an array-based menu, which pulls the menu choices from an array of strings. The array can contain any strings that are appropriate to the game system, and you can even populate the array via scripts at run-time, affording you with complete control over the options presented to the user. The array of options must be specified within the "usrArray" field, and the selected item is stored in the "usrSelect" field. If you want to display a label next to the menu, you can specify it within the "usrLabelAr" field. This behavior is only enabled when the 0th element of the "usrArray" field is non-empty.

If any of these behaviors are utilized, the selected choices are automatically integrated into the name of the pick. If you don't want the name to be changed, or if you want to synthesize your own name, you can assign the "User.NoAutoName" tag to the thing. This disables the automatic name synthesis for all picks derived from that thing.

NOTE! It is invalid to utilize more than one of the three behaviors on the same pick. It is perfectly reasonable to have one pick use a checkbox, another use a thing-based menu, a third use two thing-based menus, and a fourth use an array-based menu. However, you cannot have the same pick use a combination of behaviors, so a pick using a checkbox and a thing-based menu is not supported. If you attempt this, the results are undefined, so we recommend you not bother trying it.

REVISING OUR CODE

We need to utilize the thing-based menus for our specialized edges. To do that, our first task is to add the "UserSelect" component to the "Edge" component set. By doing that, we'll automatically add all the mechanisms for managing user-selection behaviors into every edge we define. The revised component set looks like the following.

```
<compset
```

```
id="Edge">
<compref component="Edge"/>
<compref component="MinRank"/>
<compref component="Ability"/>
<compref component="UserSelect"/>
<compref component="SpecialTab"/>
<compref component="CanAdvance"/>
</compset>
```

The next thing we need to do is add support for properly displaying our edges. We could easily utilize the built-in "UserSelect" template by swapping it into the "edEdges" table portal as the "showtemplate" in place of the "edEdge" template. However, the default display behaviors aren't exactly what we need. The defaults would work fine for the "Professional" edge, since we simply want to show a menu to select an attribute or skill. The same would also work for the similar "Expert" and "Master" edges.

The problem arises with the "Scholar" edge. The various "Knowledge" skills will often have very long names once we append the domain. A limitation of thing-based menus is that we can only show the full name of the picks within the menu. This means that we need to show two separate menus, and each needs to contain a long name (e.g. "Knowledge: Area Knowledge, Battle"). The "UserSelect" template uses normal sized menus, which use a rather large font, so it's going to be very tight trying to show two long skill names in two separate menus. Instead of having the skill names cut off, we'll switch to using a smaller menu style. This will give us plenty of space to show everything, but it requires that we not use the "UserSelect" template and implement the menus ourselves.

We don't need to display any labels next to our menus, so all we need is two menu portals. We can copy the two thing-based menu portals from the "UserSelect" template into the "edEdge" template. Once we do that, we can change the style for the portals to "menuSmall" so that we get small menus. This gives us the following two new portals.

```
<portal
  id="menu1"
  style="menuSmall">
  <menu_things
    field="usrChosen1"
    component="none"
    maxvisible="10"
    usepicksfield="usrSource1"
    candidatefield="usrCandid1">
    </menu_things>
  </portal>

<portal
  id="menu2"
  style="menuSmall">
  <menu_things
    field="usrChosen2"
    component="none"
    maxvisible="10"
    usepicksfield="usrSource2"
    candidatefield="usrCandid2">
    </menu_things>
  </portal>
```

Since the names of our picks are going to be modified to incorporate the selected options, we face the same problem we had with domains for skills. Using the "name" field for showing the name of a pick will show the modified name. This means we'll be showing the modified name and showing the menus with the same contents. That's silly, so we need to only show the base name for each pick. We accomplish that by changing the "name" portal to reference the "thingname" field.

We can now look at the contents of the Position script for the "UserSelect" template. We're going to need to determine whether our menus are visible, so we can steal that block of code. We're also going to want to highlight the menus in red if nothing is selected within them, so we can steal that code as well. We'll need to center the menus vertically, but that's nothing special. The general logic for positioning everything in the "UserSelect" template is much too complicated for our needs - we only have two menu portals - so we'll figure out how to integrate them ourselves.

When we determine the width of the "name" portal, we need to reserve some space for the menus, just in case the name is extremely long. If the name is shorter than the reserved space, we'll happily use whatever space is available. Once the name is sized properly, we can easily insert one menu into the space or split the space between two menus. Putting all this together yields the revised Position script below.

```
~set up our height based on our tallest portal
 height = portal[info].height

~if this is a "sizing" calculation, we're done
 if (issizing <> 0) then    done    endif

~determine whether our menus are visible
 ~Note: Remember that a non-empty tagexpr field indicates
menu selection is used.
 if (field[usrCandid1].isempty <> 0) then
   portal[menu1].visible = 0
   endif
 if (field[usrCandid2].isempty <> 0) then
   portal[menu2].visible = 0
   endif

~position our tallest portal at the top
 portal[info].top = 0

~position the other portals vertically
 perform portal[name].centervert
 perform portal[delete].centervert
 perform portal[menu1].centervert
 perform portal[menu2].centervert

~position the delete portal on the far right
 perform portal[delete].alignedge[right,0]

~position the info portal to the left of the delete button
 perform portal[info].alignrel[rtol,delete,-8]

~position the name on the left and use availble space, with a
gap for menus
 portal[name].left = 0
 portal[name].width =
minimum(portal[name].width,portal[info].left - portal[name].left -
150)

~position the menus to the right of the name in the available
space
 perform portal[menu1].alignrel[ltor,name,10]
 portal[menu1].width = (portal[info].left - portal[menu1].left - 20) /
2
 portal[menu2].width = portal[menu1].width
 perform portal[menu2].alignrel[ltor,menu1,10]

~if a menu is visible, make sure it has a selection
 if (portal[menu1].visible <> 0) then
   if (field[usrChosen1].ischosen = 0) then
     perform portal[menu1].setstyle[menuErrSm]
```

```
    endif
  endif
 if (portal[menu2].visible <> 0) then
  if (field[usrChosen2].ischosen = 0) then
   perform portal[menu2].setstyle[menuErrSm]
  endif
 endif
```

THE "SCHOLAR" EDGE

It's time for us to actually define one of our special edges. We'll start with the "Scholar" edge.

HOOKING UP USER-SELECTION

The first thing we need to do is hook up all the internal behaviors we need for user selection. We're going to be needing two separate thing-based menus for the two "Knowledge" skills that need to be selected. Since we only want the user to select skills that have been added to the character, we'll choose picks on the hero. This can be specified by assigning the "ChooseSrc1.Hero" and "ChooseSrc2.Hero" tags to the thing.

```
<tag group="ChooseSrc1" tag="Hero"/>
<tag group="ChooseSrc2" tag="Hero"/>
```

Each of the menus will be identical in the set of picks they show. We'll assign the same tag expression to both the "usrCandid1" and "usrCandid2" fields. We need to determine how to identify "Knowledge" skills that have a rating of d8 or higher via a tag expression. The "Knowledge" skill has a unique id of "skKnow", so every "Knowledge" skill added to the character will possess a "thingid.skKnow" tag. The die type for traits can be identified via the "trtFinal" field, which indicates half the die type value. So we need to select skills with a "trtFinal" value that is greater than or equal to 4. This yields the tag expression below.

```
thingid.skKnow & fieldval:trtFinal >= 4
```

The tag expression must be assigned to a field value. This means we need to specify the above tag expression within the "value" attribute of a "fieldval" element. However, our tag expression uses two characters that are special within XML. If this were within a PCDATA region, we could utilize a CDATA block to include the special characters freely, but this is within an attribute. Our only option is to use the proper special character sequences. This results in the two "fieldval" elements looking like the ones below.

```
 <fieldval field="usrCandid1" value="thingid.skKnow &
fieldval:trtFinal >= 4"/>
 <fieldval field="usrCandid2" value="thingid.skKnow &
fieldval:trtFinal >= 4"/>
```

REVISING THE PRE-REQUISITES

Due to the tag expression used above, the user cannot select any skills lack the necessary rating. This means the role of our pre-requisite changes a bit. First of all, we still need to recognize that the thing is invalid when the user brings up the list of edges to choose from. If there are insufficient "Knowledge" skills with a d8 rating on the character, we want to show the skill as invalid. We only do this test when we're checking the pre-requisite on a thing.

The other thing we need to handle is the case where the user changes a skill rating on us. Our menus will only allow the user to choose a skill if it satisfies the requirements. However, the user could select a skill that is valid and then change the rating on that skill after it's selected. At that point, the item is already selected and the menu won't recognize a problem. Our pre-requisite can re-verify that the two selected skills retain the minimum die type rating whenever we have a pick.

Putting these two separate tests together, we get a single pre-requisite that handles things and picks differently. But the purpose of the test remains the same, so we use the same pre-requisite to perform the test. The revised Validate script on the pre-requisite should look like the following.

```
 var total as number
  ~if this is a thing, make sure at least two skills exist with a d8
rating
  if (@ispick = 0) then
   ~go through all Knowledge skills and tally those with a d8+
rating
    foreach pick in hero where "thingid.skKnow"
     if (eachpick.field[trtFinal].value >= 4) then
      total += 1
     endif
    nexteach

 ~this is a pick, so make sure our chosen skills have a d8+ rating
  else
   if (altpick.field[usrChosen1].ischosen <> 0) then
    if (altpick.field[usrChosen1].chosen.field[trtFinal].value >= 4)
then
     total += 1
     endif
    endif
   if (altpick.field[usrChosen2].ischosen <> 0) then
    if (altpick.field[usrChosen2].chosen.field[trtFinal].value >= 4)
then
     total += 1
     endif
    endif
   endif

 ~if we have at least two valid skills, we're valid
  if (total >= 2) then
   @valid = 1
   done
   endif

 ~if we got here, we're invalid
  altthing.linkvalid = 0
```

ASSIGNING BONUSES TO SELECTED SKILLS

Our skills are now being properly selected and we can access those skills via scripts. This means that we can now automatically assign the "+2" bonus to each of those skills. We can use the "ischosen" target reference to determine if a menu has something selected. Each menu with a chosen item contains a skill that can then be accessed via the "chosen" transition. We can write a simple Eval script that applies the bonus to the selected skills, and it should look like the following.

```
 <eval index="1" phase="PreTraits" priority="5000">
   <before name="Calc trtFinal"/><![CDATA[
   ~apply the +2 modifier to each selected skill
   if (field[usrChosen1].ischosen <> 0) then
    perform
field[usrChosen1].chosen.field[trtRoll].modify[+,2,"Scholar"]
    endif
   if (field[usrChosen2].ischosen <> 0) then
    perform
field[usrChosen2].chosen.field[trtRoll].modify[+,2,"Scholar"]
```

```
    endif
  ]]></eval>
```

Custom Name

The menus are appearing properly and listing the correct skills. However, once we select skills for the edge, the name being synthesized is very unwieldy. By default, the "UserSelect" component is automatically integrating the menu selections into the name. This includes the "Knowledge" skill name, which takes up lots of room and is implied by the nature of the edge.

What we really need is a more appropriate name for our edge. Ideally, the name should just list the domains assigned to each of the "Knowledge" skills. We can accomplish this if we generate a custom name. To do that, we first have to tell the "UserSelect" component to not generate a name for us. This requires that we assign the "User.NoAutoName" tag to the thing.

```
<tag group="User" tag="NoAutoName"/>
```

We can now generate our own custom name through the use of an Eval script. If nothing is chosen from either menu, we'll include an indication that something needs to be chosen. Otherwise, we'll append only the domain names for each of the skills selected for the edge. The resulting Eval script should look like the one below.

```
<eval index="2" phase="Render" priority="5000"><![CDATA[
  ~determine the text to append to the name
  var choices as string
  if (field[usrChosen1].ischosen <> 0) then
    choices = field[usrChosen1].chosen.field[domDomain].text
  else
    choices = "-Choose-"
    endif
  if (field[usrChosen2].ischosen <> 0) then
    choices &= ", " &
field[usrChosen2].chosen.field[domDomain].text
    endif

  ~add the selection to both the livename and shortname (if
present) fields
  field[livename].text = field[thingname].text & ": " & choices
  if (tagis[component.shortname] <> 0) then
    field[shortname].text &= " (" & choices & ")"
    endif
  ]]></eval>
```

Safeguarding Against Duplicate Skills

At this point, our revised edge is working quite nicely. The menus are appearing properly and only allowing the user to select valid skills. If the skills are modified, the pre-requisite flags a suitable error. And a reasonably compact name is being synthesized for display. The only detail we haven't dealt with yet is that the user can select the same skill in each of the menus, which must be treated as an error.

We can define an Eval Rule script to detect this condition. This particular rule can only be invalid if we actually have selections in both menus. If we do, then we need to determine if the exact same pick is selected in each menu. We can't use the unique id, since every "Knowledge" skill is a separate instance of the exact same thing and will therefore have the same unique id. Fortunately, every pick within the entire portfolio (i.e. across all heroes) has a uniquely assigned index. This means we can simply check to see if the unique index values match between the two menu selections. If they do, then we have a problem. Our Eval Rule script should look like the one shown below.

```
 <evalrule index="1" phase="Validate" priority="5000"
message="Duplicate skill selected"><![CDATA[
   if (field[usrChosen1].ischosen + field[usrChosen2].ischosen <
2) then
     @valid = 1
   elseif (field[usrChosen1].chosen.uniqindex <>
field[usrChosen2].chosen.uniqindex) then
     @valid = 1
     endif
   ]]></evalrule>
```

The "Professional" Edge

We can now shift our focus to the "Professional" edge. For this edge, we need to show a menu from which the user can select a trait. We'll use the same basic approach as we used for the "Scholar" edge.

Fields and Tags

The first thing we need to do is determine what we want to show in the menu. In this case, the user can select an attribute or skill, but we only want skills that the user has added to the character. This means we'll specify the "ChooseSrc1.Hero" tag. We only need a single menu, so we can ignore adding a second tag.

Our menu needs to show traits with a minimum die-type rating of d12. Only attributes and skills can possess a die-type rating, so that means we need to include those two types of traits in the menu. We can test the field value to verify the die-type using the same technique we used for the previous edge. This results in the following field value and tag being assigned to the thing.

```
<fieldval
    field="usrCandid1"
    value="(component.Attribute | component.Skill) &
fieldval:trtFinal >= 6"/>
 <tag group="ChooseSrc1" tag="Hero"/>
```

Confer the Bonus

The next step is to confer the "+1" bonus to the selected trait. We'll use an Eval script to accomplish this, just like we did for the "Scholar" edge. The new script should look like below.

```
<eval index="1" phase="PreTraits" priority="5000">
  <before name="Calc trtFinal"/><![CDATA[
  if (field[usrChosen1].ischosen <> 0) then
    perform
field[usrChosen1].chosen.field[trtRoll].modify[+,1,"Professional"]
    endif
  ]]></eval>
```

The Pre-Requisite

The pre-requisite for the "Professional" edge will also work a lot like the one for "Scholar". When testing a thing, we need to make sure that at least one trait exists with a d12 rating. When testing a pick, we need to make sure that any selected trait still has a d12 rating. The resulting Validate script for the pre-requisite should look like the following.

```
<prereq iserror="yes" message="Any Trait of d12 required.">
  <validate><![CDATA[
    ~if this is a thing, make sure a trait exists with a d12 rating
    if (@ispick = 0) then
```

```
      foreach pick in hero where "(component.Attribute |
component.Skill)"
        if (eachpick.field[trtFinal].value >= 6) then
          @valid = 1
          done
          endif
        nexteach

    ~if a skill is chosen, make sure it has a rating of d12
     elseif (altpick.field[usrChosen1].ischosen <> 0) then
      if (altpick.field[usrChosen1].chosen.field[trtFinal].value >=
6) then
        @valid = 1
        done
        endif
      endif

    ~if we got here, we're invalid
     altthing.linkvalid = 0
     ]]></validate>
   </prereq>
```

DETECT DUPLICATES

The final task is to detect duplicates. The "Professional" edge can be selected multiple times, but a different trait must be selected every time. In order to verify this, we need to compare separate edges. One solution would be to use a "foreach" loop to scan all of the "Professional" edges and build up a list of all the selected traits, then identify any duplicates. That's a fair amount of work, and it will be even more work when we need to do the same thing for the "Expert" and "Master" edges in moment. Fortunately, there's a much easier solution we can use.

What we need to know is whether the same trait is selected multiple times by the same edge. We can detect this by assigning a tag to each trait at the same time that we confer the "+1" bonus. At the end of evaluation, we can check each trait to see if any possess two of that tag. Since we're going to be needing this same mechanism for the "Expert" and "Master" edges, we'll implement it in a way that can be easily extended for those edges.

We start by defining a new tag group, which we'll give the unique id "Duplicate". Within this group, we'll define tags for each of the edges that will require checking for duplicates. The resulting group should look like the following.

```
<group
  id="Duplicate">
  <value id="Profession"/>
  <value id="Expert"/>
  <value id="Master"/>
  </group>
```

We the assign the "Duplicate.Profession" tag to each trait that is selected for the "Professional" edge. As mentioned above, we can do this easily during the Eval script that confers the "+1" bonus by adding the line of code below.

```
perform field[usrChosen1].chosen.assign[Duplicate.Profession]
```

At this point, every trait possesses a "Duplicate" tag for each "Professional" edge that selects the trait. We can tally up the total number of "Duplicate" tags assigned to the trait. We can also tally up the number of unique "Duplicate" tags assigned to the trait, which eliminates all duplicates of the same tag. If two different "Professional" edges select the same trait, then that trait will possess two "Duplicate" tags, but it will only possess one unique "Duplicate" tag. By comparing the two quantities, we can tell if there is a problem, as shown in the new Eval Rule script below.

```
 <evalrule index="5" phase="Validate" priority="5000"
      message="Selected multiple times for same
edge"><![CDATA[
   ~if all of our tags are unique, then there are no duplicates to
report
   if (tagcount[Duplicate.?] = tagunique[Duplicate.?]) then
     @valid = 1
     done
     endif
   ]]></evalrule>
```

The above technique easily extends to the "Expert" and "Master" edges. If the trait is selected for both "Professional" and "Expert", it will have two "Duplicate" tags. However, both of those tags will be unique, so the Eval Rule will report all is well. If the trait is selected by another "Expert" edge, it will gain another tag that will not be unique and the validation error will be reported.

AVOIDING DUPLICATES

The logic above allows us to detect duplicate selections and report them as errors. It also affords us the chance to avoid duplicates during the selection of edges. Every traits that is selected by the edge is now assigned a "Duplicate" tag. We can test that condition and only show the edge as valid for selection if there are any candidate traits that don't already have the "Duplicate" tag. We can accomplish this by amending the tag expression used within the "foreach" statement in the Validate script to look like the following.

```
 foreach pick in hero where "(component.Attribute |
component.Skill) &  !Duplicate.Profession"
```

In other words, if there is one trait at d12, the "Professional" edge will appear as valid in the list of edges. Once the edge is added and that one trait selected, the edge will then appear as invalid within the list of edges. If another trait reaches the d12 rating, the edge will appear as valid again. This helps to avoid situations with duplicate selections, since the user will be warned before actually adding the edge.

THE "EXPERT" AND "MASTER" EDGES

Adding the "Expert" and "Master" edges at this point is quite easy. We'll use the exact same logic from the "Professional" edge, except for a few important tweaks.

The first change is to the tag expression within "usrCandad1" field. Since the "Expert" edge requires the "Professional" edge already be selected for the trait, we need to verify that is the case. This is done by adding a suitable term to the tag expression that checks for the "Duplicate.Profession" tag. However, we can also assume that having this tag implies that the trait is also at a d12 rating, so we can drop that requirement because it is implied. In fact, if a pick has the "Duplicate.Profession" tag, it must also be a valid trait, so we don't have to check that condition either. This leaves us with the simple requirement that the pick possess the one tag. For the "Master" edge, the same logic applies, so we only need to check for the lone "Duplicate" tag on each. The two field values end up as shown below.

```
 <fieldval field="usrCandid1" value="Duplicate.Profession"/>
```

```
<fieldval field="usrCandid1" value="Duplicate.Expert"/>
```

The second change is within the Eval script that confers the bonus. The explanation for the "modify" target reference should be changed to identify the appropriate edge by name. In addition, the appropriate "Duplicate" tag should be assigned for the edge. The revised Eval script for the "Expert" edge is shown below.

```
<eval index="1" phase="PreTraits" priority="5000">
  <before name="Calc trtFinal"/><![CDATA[
  if (field[usrChosen1].ischosen <> 0) then
    perform
field[usrChosen1].chosen.field[trtRoll].modify[+,1,"Expert"]
    perform field[usrChosen1].chosen.assign[Duplicate.Expert]
    endif
  ]]></eval>
```

There remain two final changes within the pre-requisite test. The first is within the logic used for things, wherein we can make the same assumptions about dependencies that we made above. If a trait has the "Duplicate.Profession" tag, we assume it's a valid trait, so the "foreach" statement changes to incorporate the alternate requirements, as shown below for the "Expert" edge.

```
foreach pick in hero where "Duplicate.Profession &
!Duplicate.Expert"
```

When validating a pick, we need to verify both the die-type rating and the presence of the lesser edge that is depended upon. In the case of the "Expert" edge, we need to add an additional qualification, as shown in the revised code block below.

```
if (altpick.field[usrChosen1].chosen.field[trtFinal].value >= 6)
then
  if
(altpick.field[usrChosen1].chosen.tagis[Duplicate.Profession] <>
0) then
    @valid = 1
    done
    endif
  endif
```

As long as all the equivalent changes are applied to the "Master" edge, both edges should behave exactly as they should, with all the proper dependencies and conferring the proper bonuses.

Other Character Types (NPCs and Creatures)

NPC Support (Savage)

A major benefit of HL to players is the speed and simplicity of creating characters. That same benefit should exist for GMs, but the advancement mechanism of Savage Worlds requires that characters be created one advance at a time. This leaves GMs with a choice of either creating an NPC using an incremental approach or ignoring all the validation errors by just "winging it". GMs typically need lots of NPCs, so the one-at-a-time advancement method is clunky and slow. GMs also need to have a good idea of the relative power levels of NPCs compared to the PCs, so ignoring the validation errors requires the GM to have an innate sense of how powerful a given NPC is.

There must be a better way to create NPCs using our data files. What we need is a way to create a character without any restrictions and tell the GM exactly how powerful that character is. Unless a character is created in a step-wise manner, we'll never be able to determine exactly how many advances a character used to reach a particular set of abilities. However, we can make a very close estimate, which is generally all that a GM really wants.

Our Approach

We're going to allow the GM to create NPCs freely, without any of the normal restrictions that apply to new characters. This requires that we let the user tell us whether he's creating an NPC. Since this is something fundamental to the character, we should add this to the "Configure Hero" form. This also requires that we turn off various validation tests that won't be applicable to NPCs.

Based on the various increases assigned to the character, we'll calculate how many advances were needed to reach that point. These advances must be tracked beyond the normal assignments for a starting character. Based on the advances, we can then determine how many XP were required for the character, which will then tell us the rank of the NPC.

The one main problem with this approach is that we don't know the exact order in which the advances are selected for the character. For edges, this isn't a problem, because we can simply assume that any edges satisfying its pre-requisites was taken in a valid order during the character's evolution. For attributes, this also isn't a problem, since characters are allowed exactly one attribute increase per rank. Skills are where we run into a problem with the order in which advances are taken. We can't know whether a skill was increased before or after its linked attribute was increased.

This leaves us with two options for determining the number of advances: optimism or pessimism. In an optimistic model, we assume that all skills are advanced after any linked attributes are first increased. While not always realistic, this method presumes the NPC optimized his advances to get the most out of them. The alternative is a pessimistic model, where we assume all skills are increased before their linked attributes. This approach is much less realistic than the optimistic model, so we'll go with the optimistic one.

Identifying an NPC

Before we can do anything else, we need to enable the user to identify an NPC as distinct from a normal PC. This requires that we track that state somewhere within each character. The obvious solution is to use a field on the "Actor" component, just like we did for identifying wildcards. We'll assume that all characters start out as PCs, which means our default state is for the NPC field to be zero. So we'll define a new first for the purpose, as shown below.

```
<field
  id="acIsNPC"
  name="Is NPC?"
  type="user"
  defvalue="0">
  </field>
```

While a field value is useful in most cases for identifying an NPC, there may be times we'll want to check for an NPC via a tag expression. In fact, using a tag is generally the best solution, as it's the most general technique. So we need to define a tag to identify NPCs, and we should make a point of using the tag everywhere

instead of the field for consistency. Since the tag will only ever be on the hero, we'll add it to the "Hero" tag group.

```
<value id="NPC"/>
```

We now need to assign the tag to the hero appropriately. For that, we'll define an Eval script on the "Actor" component. Within this script, we'll also take the opportunity to configure other behaviors that we want for NPCs. For example, the "Advances" tab makes zero for NPCs, so we should hide it when the character is an NPC. This results in a script like the following.

```
<eval index="6" phase="Initialize" priority="1000"><![CDATA[
  ~if this is not an NPC, just get out
  if (field[acIsNPC].value = 0) then
    done
    endif

  ~assign a tag to indicate we're an NPC
   perform hero.assign[Hero.NPC]

  ~hide components of the interface that don't apply for NPCs
   perform hero.assign[HideTab.advances]
   ]]></eval>
```

We can track whether a character is an NPC internally, so it's time to expose that the user. We decided earlier that we'll add a new option to the "Configure Hero" form (within the "cnfStart" template). We can use a simple checkbox, just like we did for the wildcard state. This checkbox will be tied to the new "acIsNPC" field and should look like the one below.

```
<portal
  id="isnpc"
  style="chkNormal"
  tiptext="Check this option to construct the character as an
unlimited NPC">
  <checkbox
    field="acIsNPC"
    message="Create Unlimited NPC?">
    </checkbox>
  </portal>
```

The next step is to place the portal appropriately within the template. However, when we take a look at the current template, there is an option in there that requires special handling. The option to specify starting XP is rather silly for an NPC. As such, we should hide that option if the user chooses to create an NPC. If we hide the portal, we then need to shift other portals around to keep everything looking good. This results in the following revised Position script for the template.

```
~set the width of the template to something we like
 width = 185

~determine whether the starting xp is visible based on if we're
an npc
 portal[xp].visible =  !hero.tagis[Hero.NPC]
 portal[lblxp].visible = portal[xp].visible

~position the title at the top
 perform portal[label].centerhorz

~position the starting cash beneath the ability slots
 perform portal[cash].alignrel[ttob,label,15]
 perform portal[lblcash].centeron[vert,cash]
 portal[cash].width = 50  portal[lblcash].left = (width -
 perform portal[cash].alignrel[ltor,lblcash,10]

~if visible, position the starting xp beneath the starting cash
 var y as number
 if (portal[xp].visible = 0) then
   y = portal[cash].bottom
 else
   perform portal[xp].alignrel[ttob,cash,15]
   perform portal[lblxp].centeron[vert,xp]
   portal[xp].width = 50
   portal[lblxp].left = (width - portal[lblxp].width - portal[xp].width -
10) / 2
   perform portal[xp].alignrel[ltor,lblxp,10]
   y = portal[xp].bottom
   endif

~position the wildcard checkbox beneath the starting xp
 perform portal[iswild].centerhorz
 portal[iswild].top = y + 15

~position the npc checkbox beneath the wildcard
 perform portal[isnpc].centerhorz
 perform portal[isnpc].alignrel[ttob,iswild,14]

~set the height of the template based on the extent of the
portals
  ~Note: Include a little extra space at the bottom for borders and
such.
  height = portal[isnpc].bottom + 3
```

This looks good, but it's a little bit tight with the menu for the specifying whether the character is an ally or enemy of the party. We can increase the gap easily within the Position script for the layout. This is accomplished by changing the top position of the "cnfAlly" portal, which results in the new line of code shown below.

```
portal[cnfAlly].top = template[cnfStart].bottom + 20
```

The final thing we need to do here is add a small safeguard. If the user enters a value for the starting XP and then selects an NPC, the portal will disappear and the character will still have a non-zero starting XP. The solution is to forcibly zero out the field if the user ever selects the NPC option. We can do this in the Eval script we added to the "Actor" component earlier by just setting the field value.

Unfortunately, the compiler complains when we try to set the field value. This is because it's a "user" field, and such should generally only be modified by the user, so an error is reported. Fortunately, we can tell the compiler we know what we're doing via use of the "trustme" statement. This results in the following lines of code being added to the end of the Eval script.

```
~force the starting XP field to zero in case the user has modified
it
  trustme
  field[acStartXP].value = 0
```

Showing Resources Differently

Once we designate a character as an NPC, we'll immediately notice a variety of behaviors that need to be modified for NPCs. The majority of these behaviors center around the way we show information to the user. For example, all the validation errors for attribute points, skill points, and edges are no longer applicable. The display of the allocation of points to those categories on the Basics

tab are now inappropriate. In addition, the title bars above attributes, skills, and edges that show the number of points left to allocate are now erroneous when we go over the starting number of points for each.

Your first thought is likely to be that we need to go through and change all of these different places to display the proper information. While that's a valid solution, it's also not the best one. The one thing that all these places have in common is that they rely on various resources that we use to track the points that are allocated by the user. If we could simply modify the way resources are handled in general, we'd be able to resolve this much more easily.

The problem with this tactic is that we use resources for multiple different situations. If we change resources in general, then we'll change them for everything. We don't want the handling of arcane powers or rewards being changed for NPCs. A character still needs the proper edges assigned to gain powers, and hindrances still need to be selected to gain offsetting rewards.

What we need is a way to customize the behavior of resources for only a specific set of resources. Fortunately, we can accomplish this without a lot of work through the use of tags. We can define a new tag that identifies a resource as being special for NPCs. Then we can assign that tag only to the individual resources that require the special handling. Within the various facets of the "Resource" component, we can check this tag and the nature of the character, using the alternate behaviors as appropriate.

We'll start by defining the new tag. Since we might find other places besides resources where we need to do something like this, we'll give it a unique id that indicates its general purpose. We'll add the tag to the "Helper" tag group, since it really doesn't belong anywhere else. The new tag will look like below.

```
<value id="NPCImpact"/>
```

Once the tag is defined, we need to put it to use. The question is which resources need to be handled specially. Looking at the nature of NPCs, there appear to be four factors that we have to handle differently. These are attributes, skills, edges, and advances. The first three of these need to be displayed differently for NPCs, so we'll identify them appropriately by assigning our new tag to each of the three things for those resources.

```
<tag group="Helper" tag="NPCImpact"/>
```

Advances are special, though. When a character is an NPC, we want the character to behave as if advances don't exist. We've already hidden the "Advances" tab panel, so we should also hide the corresponding resource from display on the "Basics" tab. We can do this by adding a ContainerReq test to the "resAdvance" thing. If the character is an NPC, then we want the thing to behave as if it doesn't exist. This results in the addition of the following to the thing.

```
<containerreq phase="Initialize" priority="2000">
  !Hero.NPC
  </containerreq>
```

We can now look at the "Resource" component and decide what changes need to be made. In all cases, we'll only leverage the alternate behavior when two separate conditions are both satisfied. First of all, the resource must possess the "Helper.NPCImpact" tag. Secondly, the character must possess the "Hero.NPC" tag. This means that we'll always use a test that looks like the following.

```
if (tagis[Helper.NPCImpact] + hero.tagis[Hero.NPC] >= 2) then
  ~use alternate behavior for NPCs
 else
  ~use normal behavior for PCs
  endif
```

Scanning through the "Resource" component from top to bottom, the first place where we'll need to tweak the behavior is in the Finalize script for the "resAddItem" field. If we have an NPC, we don't want to use any special highlighting, so we want to treat the table the same way as if all the points had been properly allocated. The code below reflects this change to the impacted lines.

```
if (tagis[Helper.NPCImpact] + hero.tagis[Hero.NPC] >= 2) then
  @text = "{text a0a0a0}"
 elseif (unspent = 0) then
```

Moving downward, the synthesis of the short name for the resource also needs to be revised specially for NPCs. If we have an NPC, all we really want to show is the number of selections made. At the top of the Eval script, we'll insert the code below to generate the text specially and just get out.

```
 ~if we have an NPC and this resource is impacted, generate
appropriate results
 if (tagis[Helper.NPCImpact] + hero.tagis[Hero.NPC] >= 2) then
  field[resShort].text = field[resSpent].value
  done
  endif
```

In the next Eval script, we're generating the summary for display, which needs to be different for NPCs. In this case, we again only want to the show the number of selections made, but we'll also append a little clarifying text after it. We'll use the same approach as above by just getting out after handling NPCs specially. This results in the new code below at the start of the Eval script.

```
 ~if we have an NPC and this resource is impacted, generate
appropriate results
 if (tagis[Helper.NPCImpact] + hero.tagis[Hero.NPC] >= 2) then
  field[resSummary].text = field[resSpent].value & " Points"
  done
  endif
```

The Eval Rule script checks for the overspending of resources, which is definitely something that we want to stop for NPCs. In this case, we simply want to treat the rule as being automatically satisfied if the character is an NPC. So we can insert the code below at the start of the script to accomplish this behavior.

```
 ~if we have an NPC and this resource is impacted, we're always
valid
 if (tagis[Helper.NPCImpact] + hero.tagis[Hero.NPC] >= 2) then
  @valid = 1
  done
  endif
```

At this point, we've now got resources properly instrumented so that specific resources will report different results appropriately for NPCs.

CALCULATING THE XP

The next thing we need to do is actually calculate the estimated XP for our NPC. Before we can do that, we need to define a new field where we can store the XP. We currently use the "resXP" resource to track the XP, but that won't work for our calculated value. So we define a new field that will be assigned the proper value, depending on whether we have an NPC or not. We'll add a Finalize script to the field that will prefix the value with "~" if we have an NPC. This will provide an reminder to the user that the value is an optimistic estimation and should be treated as such. The resulting field should look like the following.

```
<field
  id="acFinalXP"
  name="Final XP Cost"
  type="derived"
  maxfinal="10">
  <finalize><![CDATA[
   ~if the character is an NPC, indicate the value is an
approprixation
    if (hero.tagis[Hero.NPC] <> 0) then
      @text = "~" & @text
     endif
   ]]></finalize>
  </field>
```

With our field in place, we can define a new Eval script where we can do all the necessary calculations for the final XP total. At the start of the script, we'll handle the special case of a non-NPC character. For a normal character, the total XP tally is given within the "resXP" resource, so we'll pull it from there and be done.

For an NPC, there are two different factors that we need to consider. The first is the number of attribute increases. The second is the total number of advancements for the character, spanning attributes, edges, and skills.

The number of attribute increases will dictate the minimum XP level for the character. This is because only one attribute increase is allowed for each rank. If a character takes 3 attribute increases and no other advances, then it must still be a Veteran rank character and have the minimum necessary XP for that rank. It also means that the character will have extra unused advances that need to be chosen. If we have unused advances, we'll definitely want to report it to the user, so we might as well define a new field right now to store that value. This new field should look like the following.

```
<field
  id="acExtraAdv"
  name="Unused Advancements"
  type="derived">
  </field>
```

Returning to our calculation, we next calculate the total number of advances for the character. The attributes and edges are easy, since they count one apiece. Skills are where things get interesting. Since we're using an optimistic algorithm, we can iterate through each skill and determine how many advances it requires by comparing it the die type of the skill to the die type of the linked attribute. Each increment that is less than or equal to the attribute costs a half-advance (since two such skills can be raised as a single advance), while the others cost a full advance each.

Once we have the total number of advances for skills incorporated, we must adjust out the free advances we get as part of character creation. We can finally calculate the total number of XP based on the number of advances. As a last step, we then compare the minimum XP against the actual XP to determine if there are any unused advances and the final XP total to use for the character.

Putting all of this together yields the Eval script shown below. After the script is invoked, the two new fields will contain the proper values for the character.

```
 <eval index="7" phase="Final" priority="2000" name="calc
acFinalXP">
    <after name="Calc resLeft"/><![CDATA[
    ~if this is not an NPC, our final XP tally is given by the
resource
    if (hero.tagis[Hero.NPC] = 0) then
      field[acFinalXP].value = #resmax[resXP]
      done
      endif

   ~determine our minimum XP level based on the number of
attribute increases
    ~Note: If attributes are unused, set them to zero for safe use
below.
    var attribs as number
    var minxp as number
    attribs = -#resleft[resAttrib]
    if (attribs <= 0) then
      minxp = 0
      attribs = 0
    elseif (attribs <= 4) then
      minxp = (attribs - 1) * 20 + 5
    else
      minxp = 80 + (attribs - 5) * 20 + 10
      endif

   ~determine the number of advances due to attributes and
edges
    var edges as number
    var advances as number
    edges = -#resleft[resEdge]
    advances = attribs + edges

   ~go through all skills and add in the number of advances
needed for each
    ~Note: We compare the skill to the linked attribute and tally
based on that.
    foreach pick in hero where "component.Skill"
      var level as number
      var attr as number
      level = eachpick.field[trtFinal].value – 1
      attr = eachpick.linkage[attribute].field[trtFinal].value – 1
      if (level <= attr) then
        advances += level / 2
      else
        advances += attr / 2
        advances += (level - attr)
        endif
      nexteach

   ~subtract out the maximum number of skill slots allowed at
creation
    ~Note: If fewer than the allotted number of starting skill slots
are
     ~     allocated, we must not grant a refund, so limit to the
minimum.
```

```
    advances -=
minimum(#resmax[resSkill],hero.child[resSkill].field[resSpent].val
ue) / 2

   ~round the total upwards in case we've got a half-slot for a skill
used
    advances = round(advances,0,1)

   ~calculate the total number of XP necessary for all of the
advances
    var xp as number
    if (advances <= 16) then
      xp = advances * 5
    else
      xp = 80 + (advances - 16) * 10
      endif

   ~if our total xp equals or exceeds our minimum xp, setup
appropriately
    ~Note: Handle the special case of a mundane NPC that is
less than a normal PC.
    if (xp >= minxp) then
      field[acFinalXP].value = xp
      field[acExtraAdv].value = 0
    elseif (xp < 0) then
      field[acFinalXP].value = 0
      field[acExtraAdv].value = 0
    else
      field[acFinalXP].value = minxp
      if (minxp <= 80) then
        field[acExtraAdv].value = (minxp - xp) / 5
      elseif (xp >= 80) then
        field[acExtraAdv].value = (minxp - xp) / 10
      else
        field[acExtraAdv].value = (80 - xp) / 5 + (minxp - 80) / 10
        endif
      endif
    ]]></eval>
```

Integrating the XP

The final XP tally is now stored in the new "acFinalXP" field, but nothing is using it. We need to switch all the appropriate places over to use the new field. All references to the XP should be using the "resXP" resource within our data files, so we should be able to do a search through the files to identify uses of "resXP". We can then assess which ones need to be revised to use the new field.

The first instance we need to change is in the Calculate script for the "acRank" field of the "Actor" component. In addition to changing the field reference, we also need to modify the timing of the script. The new Eval script we defined to calculate the XP is assigned a priority of 1000. Since this script relies on that value, we need to make sure it occurs after the value is in place. For safety, we'll also add a timing dependency on the other script so that any future timing changes will be verified by the compiler. This results in a new Calculate script similar to the following.

```
 <calculate phase="Final" priority="5000">
    <after name="calc acFinalXP"/><![CDATA[
    var xp as number
    xp = field[acFinalXP].value
    if (xp < 80) then
      @value = round(xp / 20,0,-1)
    else
      @value = 4
      endif
    ]]></calculate>
```

The next use we come across that requires change is the Eval script that generates the recap summary for allies. We can simply change the one line that references the resource to the new field, which should look like below.

```
recap &= ranktext & "  (" & field[acFinalXP].value & " XP)"
```

There are two tab panels where the XP is shown to the user, and both can be changed very simply to the new field. On the "Basics" tab, the Label script of the "baRank" portal needs one line changed, which should look like below.

```
@text = ranktext & "  (" & herofield[acFinalXP].text & " XP)"
```

Similarly, one the "Journal" tab, a single line of the Label script of the "info" portal must be changed to the following.

```
@text &= "{horz 40} Total XP: " & herofield[acFinalXP].text
```

The remaining two instances that need to be modified are within the character sheet. On the first page of the character sheet, the Label script of the "oHeroInfo" portal should be changed to reference the new field. The same basic change must be made within the "details" portal of the "oAllyPick" template on the second page.

Reporting Inconsistencies

When we calculated the XP for NPCs, we identified a potential problem that needs to be reported to users as a validation error. If a character takes multiple attribute increases, it's possible that not enough other advances will be taken to fill in the gaps. We have this state already recorded in a field, so all we need to do is test is somewhere.

The best solution is to add a new thing specifically for validation of NPCs. That way, we can ensure that problem is automatically associated with the nature of the character being an NPC. We can then write a single Eval Rule for the thing that tests the field and reports a suitable error if there is a problem. Our new thing should be added to "thing_validate.dat" and should look like the one below.

```
 <thing
    id="valNPC"
    name="NPC"
    compset="Simple">
    <tag group="Helper" tag="Bootstrap"/>

   <evalrule index="1" phase="Validate" priority="8000"
        message="???" summary="Unused
Advances"><![CDATA[
      ~if we have zero unused advances, we're good
      if (herofield[acExtraAdv].value = 0) then
        @valid = 1
        done
       endif

     ~synthesize a validation message with the number of unused
advances
      @message = "Selected attributes confer " &
herofield[acExtraAdv].value & " unused advance(s)   to allocate"

     ~mark associated tabs as invalid
      container.panelvalid[skills] = 0
      container.panelvalid[edges] = 0
      ]]></evalrule>
   </thing>
```

TESTING EVERYTHING

We can now go through everything and test how it all works. There don't appear to be any problems until we print out a character sheet for an NPC. We get an error that we're trying to access the non-live pick "resAdvance". That's the resource that we assigned the ContainerReq test, so somewhere our code is assuming that the "resAdvance" pick can be accessed all the time.

If we do a quick search for "resAdvance" within the data files associated with character sheet output, we can spot the problem pretty quickly. In the "oHeroInfo" portal on the first page, any unused advances are being reported. Since the pick is non-live for NPCs, the error is reported. We can easily add a test to ensure that we only append advances when the "resAdvance" pick is live. The revised code should look like below.

```
if (hero.childlives[resAdvance] <> 0) then
  if (#resleft[resAdvance] > 0) then
    @text &= "; " & #resleft[resAdvance] & " Advance(s)"
    endif
  endif
```

But wait a minute. This is a golden opportunity for us to include an indication of unused advances for NPCs as well. We have the "acExtraAdv" variable to tell us if there are any extra advances, so we might as well use it here. We'll change the code to use this alternate field for NPCs, which results in the following revised code.

```
~report any unspent advances
 ~Note: We have to handle NPCs and non-NPCs differently.
 var unused as number
  if (hero.tagis[Hero.NPC] = 0) then
   unused = #resleft[resAdvance]
  else
   unused = herofield[acExtraAdv].value
   endif
  if (unused > 0) then
   @text &= "; " & unused & " Advance(s)"
  endif
```

MORE CLEANUP (SAVAGE)

Many of the tasks we came up with previously have been addressed. Let's take this opportunity to do another round of testing everything and see if there's anything we missed last time. Then we can knock off most of the remaining little things on our list. The sections below identify the various issues we can spot and the fixes that we need to make.

DAMAGE AND NON-WILDCARDS

On the "In-Play" tab, we make the assumption that all characters can take four levels of damage. However, this only applies to wildcards. Non-wildcards take one hit and go down. After looking into this a little bit further, it seems that we've made this assumption through the data files. We need to properly handle the situation where a character has a different maximum number of wounds based on the wildcard designation.

The first thing we need to do is setup a new field into which we can save the appropriate maximum number of wounds. Since the field value is based solely on the wildcard state of the character, we can use a Calculate script on the field to easily determine the value. We need to be sure to schedule the script early, since other scripts will rely on this value. This yields a field that looks like the following.

```
<field
  id="acMaxWound"
  name="Maximum Wounds"
  type="derived">
  <calculate phase="Setup" priority="1000"><![CDATA[
   ~if we're a wildcard, we can take 4 wounds, else only one
   if (field[acIsWild].value <> 0) then
     @value = 4
   else
     @value = 1
    endif
  ]]></calculate>
 </field>
```

With the field in place, we can now go through the data files and identify places where the field needs to be utilized. We'll do a search for references to the "acWounds" field to locate places where a change may be needed. It turns out there are three places within the "Actor" component and another two places within the "In-Play" tab.

The Finalize script for the "acDmgSumm" field uses a hard-coded value of four when determining whether to show the wounds as "Inc". We can replace the four with a reference to the "acMaxWound" field. The new line of code should look like below.

```
if (wounds <= -field[acMaxWound].value) then
```

Similar logic is utilized within the Finalize script for the "acDmgTac" field. Again, we replace the four with a reference to the field. The third reference is in the Eval script that assigns the "Hero.Dead" tag. Simply swapping out the literal value for the field is all we need to do. Shifting our focus to the "In-Play" tab, both the "wounds" and "wndsustain" portals reference a hard-coded value of four. Both of these references need to be replaced with the field. The new line of code for the Label script in the "wounds" portal is shown below.

```
if (wounds > -field[acMaxWound].value) then
```

All of the handling for wounds is now properly differentiating between wildcards and non-wildcards.

LEAD SUMMARY SCRIPT

When a lead character is saved out to a portfolio, a summary of that character is synthesized and stored in the portfolio. This summary is displayed along with the name when the user attempts to import characters from that portfolio. The summary is synthesized via the LoadSummary script, whose purpose is to provide a brief (but useful) synopsis of the character.

The LoadSummary script is defined in the file "definition.def", and the Skeleton files provide us with a basic implementation. We'll adapt the one provided for Savage Worlds. The question is what information we should include. Savage Worlds characters have no clear role (such as classes), so there's really nothing definitive we can say about a character beyond its race and rank/XP. This leaves us with a script implementation that looks like the following.

```
~start with the race
 var txt as string
 txt = hero.firstchild["Race.?"].field[name].text
if (empty(txt) <> 0) then
   txt = "No Race"
```

```
  endif
@text &= txt

~append the rank and XP
 var rankvalue as number
var ranktext as string
 rankvalue = herofield[acRank].value
 call RankName
 @text &= " - " & ranktext & " - " & herofield[acFinalXP].value & "
XP"
```

Arcane Power Totals

If a character with an arcane background adds a foreign gizmo on the "Gear" tab, that gizmo is being counted towards the total number of arcane powers selected by that character. Foreign gizmos need to be ignored when tallying up the number of powers chosen for the character.

The tallying of each arcane power is performed within an Eval script on the "Power" copmonent. This script always consumes one power slot for each power. We need to modify the script to skip any power with the "Helper.Foreign" tag. The revised script should look like below.

```
if (tagis[Helper.Foreign] = 0) then
  perform #resspent[resPowers,+,1,field[name].text]
  endif
```

Show Arcane Powers Resource

The "Basics" tab contains all of the various resources that a user will want to monitor during character creation, except for one. If the character has an arcane background, the list of resources does not include the one for arcane powers.

We can add the resource to the list by assigning it the "Helper.Creation" tag. We can then control its position in the list by assigning it an appropriate "explicit" tag. We'll put it at the end, after "Skills", so we'll assign it the tag "explicit.6".

The only problem now is that the resource shows up for characters with no arcane background. What we want is for the resource to only show up when an arcane background is possessed. There are a number of ways we can solve this, but the best is to recognize that the resource serves no purpose unless we an arcane background is selected. As such, we can use the same approach as with the "Advances" resource, except that pre-condition changes. We can assign a ContainerReq test to the "resPowers" resource and make the thing dependent upon the hero possessing any tag from the "Arcane" group, which will assigned if any arcane background is chosen. This results in the revised "resPowers" thing below.

```
<thing
  id="resPowers"
  name="Arcane Powers"
  compset="Resource">
  <fieldval field="resObject" value="Arcane Power"/>
  <tag group="Helper" tag="Bootstrap"/>
  <tag group="Helper" tag="Creation"/>
  <tag group="explicit" tag="6"/>

 <containerreq phase="Initialize" priority="2000">
   Arcane.?
   </containerreq>
  </thing>
```

Starting Bennies

Normal player characters receive three Bennies at the start of each game. However, NPCs that are wildcards only receive two personal Bennies and non-wildcards receive zero. We have the necessary information to setup everything properly, so let's add this in properly.

Bennies are managed via a tracker that is defined in the file "thing_miscellaneous.dat". This tracker starts with a fixed range of zero to three. By defining an Eval script for the thing, we can customize the starting values based on the characteristics assigned to the hero. The new Eval script should look similar to the one below.

```
<eval index="1" phase="Initialize" priority="6000"><![CDATA[
  ~if we're not a wildcard, we get zero starting Bennies
  if (herofield[acIsWild].value = 0) then
    field[trkMax].value = 0

  ~if we're an NPC, we get two starting Bennies
  elseif (hero.tagis[Hero.NPC] <> 0) then
    field[trkMax].value = 2

  ~otherwise, we're a normal character and get three starting
Bennies
  else
    field[trkMax].value = 3
    endif
  ]]></eval>
```

Super Power Skill Names

The character sheet output makes the assumption that all skills that lack a domain can safely fit within the narrow width of the two-column table. While this is true for all normal skills, the skills associated with super powers have comparatively much longer names. As such, they are shrunk as far as possible and still cut off.

A simple solution is to handle super power skills the same way we handle skills with domains, moving them into the single-column table beneath the two-column table. Since all super skills possess an "Arcane.?" tag, we can readily identify such skills. This makes it easy to integrate them into the second table instead.

The first step is to revise the List tag expression for the skills shown in the two-column table, omitting the super skills. The second step is to revise the List tag expression of the other table to include the super skills. The two revised List tag expressions should look like below. Note the addition of the "CDATA" block around the first one due to the use of the '&' character.

```
<list><![CDATA[!User.NeedDomain &  !Arcane.?]]></list>

<list>User.NeedDomain | Arcane.?</list>
```

Once the tag expressions are defined, we then need to refine the "oSkillPick" template. The template bases the sizes for its contents based on the same criteria used in the List tag expressions for the tables. This means all we need to do is change the one line in the Position script that identifies whether the pick belongs in the narrow or wide table. The revised line of code should look like the following.

```
if (tagis[User.NeedDomain] + tagis[Arcane.?] = 0) then
```

ALTERNATE PACE VALUES

There are some abilities that necessitate having two values for the "Pace" trait. All characters possess a ground-based speed, which is the standard value for "Pace". However, some abilities confer swimming or flying speeds. We need a way to track and convey that information to the user.

The easiest solution is to add a new field to the "Derived" component. This one field will serve to track a generic alternate value, and we can integrate the field value into the display for the trait. If there is no alternate value needed, then it can be left at a default value of zero, in which case it will be omitted from display. For the "Pace" trait, this field will identify whichever alternate speed is appropriate (flying or swimming). There is currently no need for this field on other derived traits.

The new field is extremely simple. We'll designate it as a "special" value, since it's actual purpose will value from one trait to another. The definition below is all we need.

```
<field
  id="trtSpecial"
  name="Special Value"
  type="derived">
  </field>
```

With the field defined, we then need to decide how to integrate it into the displayed value for the trait, which is stored in the "trtDisplay" field. An Eval script on the "Trait" component currently handles this, so we'll need to break up the logic. We'll modify the existing Eval script on the "Trait" component to do nothing for a "Derived" trait. Then we'll add a new Eval script on the "Derived" component to synthesize the proper display contents. The new Eval script should look like the following.

```
 <eval index="2" phase="Render" priority="5000" name="Calc
trtDisplay"><![CDATA[
   ~our display text is the final value, plus any "special" value
given
   field[trtDisplay].text = field[trtFinal].value
   if (field[trtSpecial].value <> 0) then
     field[trtDisplay].text &= "/" & field[trtSpecial].value
     endif
   ]]></eval>
```

If a particular ability confers a non-standard value for "Pace", it can assign that value to the "trtSpecial" field. That field will then be integrated into the displayed value for the trait.

NATURAL ARMOR

The Skeleton files provide an entry for "Natural Armor". This thing can be used for any situation where an actor has an innate (or natural) defensive protection that behaves like armor. For example, an extra thick hide or scaly skin might confer the equivalent defense of armor.

When this situation presents itself, you can bootstrap the "armNatural" thing onto the character. As part of the bootstrap, you can specify the level of defensive protection that is afforded by the natural armor via the "defDefense" field. If not specified, the default defensive rating is "1". The sample "bootstrap" element below shows the assignment of natural armor with a defensive rating of "3".

```
<bootstrap thing="armNatural">
  <assignval field="defDefense" value="2"/>
  </bootstrap>
```

Since natural armor covers the entire body, we need to assign all of the various "ArmorLoc" tags to the thing. We also need to assign an appropriate armor type, which means we need to define an appropriate tag in the "ArmorType" group. When we're done, the revised thing should look like below.

```
<thing
  id="armNatural"
  name="Natural Armor"
  compset="Armor"
  description="Description goes here"
  isunique="yes"
  holdable="no">
  <fieldval field="defDefense" value="1"/>
  <tag group="Equipment" tag="Natural"/>
  <tag group="Equipment" tag="AutoEquip"/>
  <tag group="ArmorType" tag="Natural"/>
  <tag group="ArmorLoc" tag="Torso"/>
  <tag group="ArmorLoc" tag="Arms"/>
  <tag group="ArmorLoc" tag="Legs"/>
  <tag group="ArmorLoc" tag="Head"/>
  </thing>
```

REVISE "CONFIGURE HERO" FORM

The "Configure Hero" form currently shows the starting cash, followed by the starting XP, and then the checkboxes to designate a wildcard and an NPC. The result is a sequence where toggling the NPC checkbox causes the portals for starting XP to appear and disappear above it. This behavior is rather disorienting to the user, as the checkbox portal just toggled moves under the mouse.

What we need is a sequence that allows the visibility change in an intuitive manner. That sequence should be the NPC checkbox, followed by the wildcard checkbox, then the starting cash, and lastly the starting XP. Since the concept of starting cash really doesn't make any sense for NPCs, we'll also hide the starting cash for an NPC.

We'll first re-order the portals within the template to this new sequence. This will ensure that attempts by the user to use the <Tab> key to move through the portals will work smoothly.

We now need to revise the Position script logic to orchestrate the display properly. We start by determining the visibility of the starting cash and XP. After that, we'll position the label at the top, with the NPC checkbox beneath it. We'll allow for the remaining portals to be optionally visible, tracking our vertical position as we proceed downward through the template. This results in the following revised Position script.

```
 ~set the width of the template to something we like
  width = 185

 ~determine whether the starting cash is visible based on
whether we're a pc
  portal[cash].visible =  !hero.tagis[Hero.NPC]
  portal[lblcash].visible = portal[cash].visible
```

```
 ~determine whether the starting xp is visible based on if we're a
pc
  portal[xp].visible =  !hero.tagis[Hero.NPC]
  portal[lblxp].visible = portal[xp].visible

 ~position the title at the top
  perform portal[label].centerhorz

 ~position the npc checkbox beneath the title
  perform portal[isnpc].centerhorz
  perform portal[isnpc].alignrel[ttob,label,15]

 ~start tracking our vertical position because portals may not be
visible
  var y as number
  y = portal[isnpc].bottom

 ~if visible, position the wildcard checkbox beneath the character
type
  if (portal[iswild].visible <> 0) then
    perform portal[iswild].centerhorz
    portal[iswild].top = y + 15
    y = portal[iswild].bottom
    endif

 ~if visible, position the starting cash next
  if (portal[cash].visible <> 0) then
    portal[cash].top = y + 15
    portal[cash].width = 50
    perform portal[lblcash].centeron[vert,cash]
    portal[lblcash].left = (width - portal[lblcash].width -
portal[cash].width - 10) / 2
    perform portal[cash].alignrel[ltor,lblcash,10]
    y = portal[cash].bottom
    endif

 ~if visible, position the starting xp beneath the wildcard
checkbox
  if (portal[xp].visible <> 0) then
    portal[xp].top = y + 15
    portal[xp].width = 50
    perform portal[lblxp].centeron[vert,xp]
    portal[lblxp].left = (width - portal[lblxp].width - portal[xp].width -
10) / 2
    perform portal[xp].alignrel[ltor,lblxp,10]
    y = portal[xp].bottom
    endif

 ~set the height of the template based on the extent of the
portals
  ~Note: Include a little extra space at the bottom for borders and
such.
  height = y + 3
```

The final thing we'll do is change the label at the top of the template. It currently says "Starting Resources", but the NPC state is not a resource, so the name is misleading. We'll change it to "Starting Characteristics". This requires simply changing the literal text for the "label" portal, and we're done.

CUSTOMIZABLE ABILITIES

There are no races actually defined in the core rulebook (other than Humans). However, there are many races defined in the various supplements. Abilities will also be used with all the various creatures throughout the core rulebook and supplements. Many of the special abilities are basically the same thing with slight variations. For example, different races and creatures possess natural weapons and/or natural armor.

The names and bonuses of these special abilities often differ, but the mechanics are all the same. This results in the need to define lots of very similar abilities. It would be much easier if we simply provided some customizable special abilities that could be easily re-used and adapted.

Let's take a look at the types of customization that we need. For natural armor, we need to modify the name (e.g. "Carapace") and specify the numeric armor rating to be used (e.g. "2"). If we scan through the various creature abilities, the "Size" ability requires a numeric rating, so it can be handled the same way as natural armor. For natural weapons, we need to modify the name (e.g. "Bite", "Claw", etc.) and specify the appropriate damage the attack. This may need to be done as a weapon die or as text for odd situations (e.g. "2d6"). We also need to handle situation like weaknesses and immunities for creatures, which need to specify custom text with the details.

This leaves us with the need for three separate fields that can be used to customize a special ability. For the weapon die, we'll use the existing tag mechanism that is already in place. We can already modify the name via use of the "livename" field. So we really only need two new customization fields - a value and a string. We'll add these to the "RaceAbil" component as shown below.

```
<field
  id="abilValue"
  name="Extra Value"
  type="derived">
  </field>

<field
  id="abilText"
  name="Extra Text"
  type="derived"
  maxlength="25">
  </field>
```

We can now put these fields to use by defining fully customizable versions of natural armor and weapons. We'll start with the natural armor, which we'll define as racial ability. There is already an existing "Natural Armor" piece of armor, so we're defining the corresponding special ability here. Our ability must bootstrap the actual armor. Then we can take the value assigned to the ability and use it to set the defensive rating of the armor, as well as customize the name of the ability. This yields the following ability.

```
<thing
  id="abArmor"
  name="Natural Armor"
  compset="RaceAbil"
  isunique="yes"
  description="">
  <bootstrap thing="armNatural"/>
  <eval index="1" phase="Effects" priority="5000">
    <before name="Calc trtFinal"/><![CDATA[
    ~set the defensive value for the armor
    perform hero.child[armNatural].setfocus
    focus.field[defDefense].value = field[abilValue].value

   ~if we've been assigned a custom name, assign it to the
armor
    if (empty(field[livename].text) = 0) then
      focus.field[livename].text = field[livename].text
      endif

   ~append the value to both names
    field[livename].text = field[name].text & " " &
signed(field[abilValue].value)
```

```
    focus.field[livename].text = focus.field[name].text & " " &
signed(field[abilValue].value)
    ]]></eval>
  </thing>
```

Our new ability can be both used and customized by a race quite easily. The ability is bootstrapped onto the actor, and the appropriate fields are assigned as part of the bootstrap. The "abilValue" field dictates the armor rating to be used. If the ability name also needs to be customized, the "livename" field can specified. An example of this is shown below for natural armor that is called a "Carapace" and confers a "+2" defensive bonus.

```
<bootstrap thing="abArmor">
  <assignval field="livename" value="Carapace"/>
  <assignval field="abilValue" value="2"/>
  </bootstrap>
```

We can define a similar special ability for use as a natural weapon. The "Unarmed Attack" weapon is always bootstrapped to every actor, so we don't need to bootstrap anything here. Instead, we simply customize the already existing weapon to suit our needs. The damage associated with the natural weapon can be specified in two different ways. In general, it will be assigned via the appropriate "WeaponDie" tag, but it can also be specified via the "abilText" field. The damage will be appended to both the weapon damage and the name of the ability. We handle it this way so that the ability name shows the extra damage, but the weapon entry itself omits the extra damage from the name.

There is a critical detail we have to be careful about here. The "WeaponDie" tag is used at a timing of Initialize/5000 to setup the damage properly for the weapon. This means that we must forward the tag to the "Unarmed Attack" weapon before then. However, all of the other logic should be performed later, such as within the Effects phase. Consequently, we need to define two separate Eval scripts, with each doing different pieces of the handling. This yields the following special ability.

```
<thing
  id="abWeapon"
  name="Weapon"
  compset="RaceAbil"
  isunique="yes"
  description="">
  <eval index="1" phase="Initialize" priority="2000"><![CDATA[
    ~if we have a weapon die, we must forward it immediately
(before it's used)
    if (tagis[WeaponDie.?] <> 0) then
      perform hero.child[wpUnarmed].pushtags[WeaponDie.?]
      endif
    ]]></eval>
  <eval index="2" phase="Effects" priority="5000"><![CDATA[
    ~customize the Unarmed Attack pick with the appropriate
name
    ~Note: Do this BEFORE modifying our name below to use
an undecorated name.
    perform hero.child[wpUnarmed].setfocus
    focus.field[livename].text = field[name].text

    ~get our ability text; if we have none, assume "Str"
    ~Note: This handles the case of a natural weapon with no
additional die.
    var ability as string
    ability = field[abilText].text
    if (empty(ability) <> 0) then
      ability = "Str"
    endif
    ~if we don't have a weapon die, assign the damage text
    if (tagis[WeaponDie.?] = 0) then
      focus.field[wpDamage].text = ability
      endif

    ~if our live name is currently empty, use our default name
    if (field[livename].isempty <> 0) then
      field[livename].text = field[name].text
      endif

    ~update our name appropriately for display
    if (tagis[WeaponDie.?] <> 0) then
      var die as number
      die = tagvalue[WeaponDie.?] * 2
      field[livename].text &= " (Str+d" & die & ")"
    else
      field[livename].text &= " (" & ability & ")"
      endif
    ]]></eval>
  </thing>
```

Putting our new special ability to use is very simple. Using the "WeaponDie" method, we bootstrap the ability onto the actor, customize "livename" field, and assign the proper tag. The example below shows a natural weapon that is called a "Bite" and does "Str+d6" damage.

```
<bootstrap thing="abWeapon">
  <autotag group="WeaponDie" tag="3"/>
  <assignval field="livename" value="Bite"/>
  </bootstrap>
```

If the damage were something non-standard, we would use the alternate method. We would bootstrap the ability onto the actor and customize both the "abilText" and "livename" fields. The example below shows a natural weapon that is called a "Non-Standard" and confers "2d6" damage.

```
<bootstrap thing="abWeapon">
  <assignval field="livename" value="Non-Standard"/>
  <assignval field="abilText" value="2d6"/>
  </bootstrap>
```

VIGOR AND FIGHTING BEYOND "D12"

One of the things we deferred early on is the special handling needed for a couple of derived traits. If the "Vigor" attribute is greater than a "d12" rating, we need to factor that into the "Toughness" trait calculation. Similarly, if the "Fighting" skill exceeds "d12", we need to factor that into the "Parry" trait.

For the "Toughness" trait, we can revise the Eval script that calculates the trait bonus. If the trait is at a "d12" rating (i.e. a value of 6), then we add half the "trtRoll" field value. Since the rules call for the bonus to be increased by "+1" for every two full points, we need to divide by two and round the value down. This yields the revised Eval script shown below.

```
~toughness is 2 plus half the character's Vigor, but we track
attributes at
 ~the half value (2-6), so we add Vigor directly; we get the Vigor
by using
 ~the "#trait" macro
 ~Note: We must also handle an overage beyond d12 on a +1
per 2 full points basis.
```

```
  ~Note: We ADD the amount in case other effects have already
applied adjustments.
  var bonus as number
  bonus = #trait[attrVig]
  if (bonus >= 6) then
    bonus += round(hero.child[attrVig].field[trtRoll].value / 2,0,-1)
    endif
  perform field[trtBonus].modify[+,bonus,"Half Vigor"]
  ~equipped armor should add to the Toughness, so we add that
from the armor
```

We can apply the exact same logic to the "Parry" trait. This yields the revised Eval script below.

```
 ~parry is 2 plus half the Fighting skill, but we track all skills at
the half
  ~value (2-6), so we add the Fighting skill; since the skill might
not exist
  ~for the character, we use the "#traitfound" macro, which
returns the value
  ~of the trait if found and zero if not
  ~Note: We must also handle an overage beyond d12 on a +1
per 2 full points basis.
  ~Note: We ADD the amount in case other effects have already
applied adjustments.
  if (hero.childexists[skFighting] <> 0) then
    var bonus as number
    bonus = #trait[skFighting]
    if (bonus >= 6) then
      bonus += round(hero.child[skFighting].field[trtRoll].value /
2,0,-1)
      endif
    perform field[trtBonus].modify[+,bonus,"Half Fighting"]
    endif
```

Bounding of Traits

The contents of the "trtUser" field for each trait are bounded. For attributes and skills, the bounding applies a minimum value of "2" and a maximum of "6". These values correspond to the die-types "d4" through "d12".

Under normal conditions, these bounds work great. The problem arises when effects are applied that adjust the traits via the "trtBonus" field. For example, the "Dwarven" race possesses the "Tough" special ability that confers a free extra die on the "Vigor" trait. If the "Dwarven" race is selected, the starting "d4" in "Vigor" changes to "d6", just as it should. Increasing one notch goes to "d8", then to "d10", and then to "d12". When we reach "d12", we should not be allowed to go any higher, but we're allowed to increment again, which does nothing but consume an attribute point for no benefit.

The reason for this behavior is that we are bounding the "trtUser" field value alone. We are not taking into consideration any adjustment from the "trtBonus" field value. If the "trtBonus" value is greater than zero, we need to decrement the maximum by the corresponding amount. This ensures we stop the user from increasing the trait as soon as it reaches the maximum of "d12". The revised Bound script for the field should look like below.

```
@minimum = field[trtMinimum].value
 @maximum = field[trtMaximum].value
 if (field[trtBonus].value > 0) then
   @maximum -= field[trtBonus].value
   endif
```

Adjustments Require Hiding

All of our permanent and in-play adjustments appear to be working correctly. However, the list of traits shown in the menu includes choices that should not be visible. For example, the list of attributes show our special attribute for super powers that should never be visible to the user.

The problem is that the tag expressions used on the various adjustments don't include handling for the hidden traits. The tag expression for the attribute die adjustment simply specifies the "Attribute" component. It also needs to to omit any attributes that possess the "Hide.Attribute" tag. This same basic change is needed on a total of five of the adjustments.

The "adjAttrD" and "adjAttrR" adjustments both need to omit attributes with the "Hide.Attribute" tag. The revised "adjCandid" field for both of these things should look like below.

```
 <fieldval field="adjCandid" value="component.Attribute &
!Hide.Attribute"/>
```

Similarly, the "adjSkillD" and "adjSkillRS" adjustments both need to omit skills with the "Hide.Skill" tag. The revised "adjCandid" field for both of these things should look like the following.

```
 <fieldval field="adjCandid" value="component.Skill &
!Hide.Skill"/>
```

Lastly, the "adjDerived" adjustment needs to omit traits with the "Hide.Trait" tag. The revised "adjCandid" field for this thing should be as follows.

```
 <fieldval field="adjCandid" value="component.Derived &
!Hide.Trait"/>
```

Centralize Rank Name Handling

There are more than a half-dozen places where the "RankName" procedure is called to convert the rank as a value to the corresponding name for display. In all but one of these places, the rank value being used is the rank of the current actor. There is no reason for us to go through the logic of setting up to call the procedure each time.

Instead, we can simplify everything by adding a new field to the "Actor" component where we can synthesize and store the rank name a single time. Then we can have each of these places simply retrieve the field from the hero. Our new field should look similar to the one below.

```
<field
  id="acRankName"
  name="Current Rank as Name"
  type="derived"
  maxlength="15">
  </field>
```

We can then add a new Eval script to the component that will synthesize and save the name properly. We need to schedule this script after the value is determine, and it must be done before we need to reference the field. The new script should look like below.

```
<eval index="8" phase="Render" priority="1000"><![CDATA[
```

```
    ~convert our current rank value to the corresponding name
and save it
    var rankvalue as number
    var ranktext as string
    rankvalue = field[acRank].value
    call RankName
    field[acRankName].text = ranktext
    ]]></eval>
```

A suitable field is now in place on the actor. We can go through all calls to the "RankName" procedure and change most of them to retrieve the new field instead of calling the procedure itself. The first instance is in the "Actor" component itself, within the Eval script that synthesizes the recap summary for allies. We can replace the code that invokes the procedure with a single line, as shown below.

```
 recap &= field[acRankName].text & "  (" & field[acFinalXP].value
& " XP)
```

The next instance is in the LeadSummary script within the definition file. As we did above, we can replace the block of code with a single line shown below.

```
 @text &= " - " & herofield[acRankName].text & " - " &
herofield[acFinalXP].value & " XP"
```

The instance we come across is in the "MinRank" component, where the rank is referenced within the pre-requisite test. This time, we are determining the displayed rank based on the requirement specified instead of the actual rank of the actor. So there is nothing to change here.

There are two separate tabs where the rank name is displayed, and both need to be changed. On the "Basics" tab, the "baRank" portal can be replaced with a single line of code, just like we did above. On the "Journal" tab, the synthesis of the info shown across the top and eliminate the procedure call and simply reference the field.

The remaining two places are within the character sheet output. Both on the first sheet and the second sheet, the rank name is included in the output. As we did above, both instances can be swapped out for a single line of code.

CREATURES (SAVAGE WORLDS)

If we want to be big help for GMs, we need to allow GMs to do more than just create NPCs. We must also let them quickly create monsters and animals. Since both need to be handled the same way, we'll refer to them generically as "creatures".

NEW CHARACTER TYPE

Since creatures are fully defined within the rulebook, there is no substantial customization to be done, and there are definitely no rules that govern construction of creatures. Consequently, the first thing we need to do is add support for a new character type - the creature.

We currently has a field that indicates whether the character is an NPC or not. Since we'll now have three types of character, we'll change the field for more generalized use. The "acIsNPC" field can be changed to an "acCharType" field, where the value of the field can be one of three possibilities. A "zero" indicates a normal PC, a "one" indicates an NPC, and a "two" indicates a creature. The new field will look like below.

```
<field
  id="acCharType"
  name="Character Type"
  type="user"
  defvalue="0">
  </field>
```

We have a tag that identifies an NPC, but we really need separate tags to indicate all three different character types. We'll assign the appropriate tag based on the field value. The new tags will be defined within the "Hero" tag group and should include the following.

```
<value id="PC"/>
 <value id="Creature"/>
```

The existing Eval script that identifies and NPC and sets up appropriately can be readily adapted. Instead of just recognizing an NPC, the script can recognize all three different values within the "acCharType" field. Based on the field value, the appropriate tag can be assigned to the hero, plus any other customizing of the interface. For example, the "Journal" tab is of no use for a creature. We're also going to need to either overhaul the "Edges" tab to only show racial abilities or replace it with an alternate tab. We'll assume the latter for now, which results in the following revised Eval script on the "Actor" component.

```
 ~assign a tag to indicate we're a PC, NPC, or Creature, as
appropriate
  if (field[acCharType].value = 0) then
    perform hero.assign[Hero.PC]
  elseif (field[acCharType].value = 1) then
    perform hero.assign[Hero.NPC]  else
    perform hero.assign[Hero.Creature]
    endif

 ~if this is a standard PC, there's nothing more to do  if
(hero.tagis[Hero.PC] <> 0) then
    done
    endif

 ~hide components of the interface that don't apply for NPCs
and/or Creatures
  perform hero.assign[HideTab.advances]  if
(hero.tagis[Hero.Creature] <> 0) then
    perform hero.assign[HideTab.edges]
    perform hero.assign[HideTab.journal]
    endif

 ~force the starting XP field to zero in case the user has modified
it
  trustme
  field[acStartXP].value = 0
```

INTEGRATE INTO CONFIGURE HERO FORM

Now that we've changed the field, we need to change the "Configure Hero" form to properly set the field. We currently have a checkbox to designate whether the character is an NPC. We need to replace this checkbox with a way to choose between the three options: PC, NPC, and creature.

The easiest way to do this is with a menu that consists of literal choices. That way, we don't have to go through the extra work of defining an assortment of "things" to represent each option. With a literal menu, we have complete control over the choices presented

and the corresponding values associated with each choice. We just need to spell out the list of choices as part of the menu definition.

You'll find an example of using a menu like this on the "Personal" tab. It's the one for selecting the character's gender, with options for male and female. We can copy this menu and adapt it for our needs. We've already determined the meaning of the values 0-2 for the "acCharType" field, so all we need to do is define corresponding options. This results in a new menu portal that looks like below.

```
<portal
  id="chartype"
  style="menuNormal">
  <menu_literal
    field="acCharType">
      <choice value="0" display="Type: Player Character"/>
      <choice value="1" display="Type: NPC (Unlimited)"/>
      <choice value="2" display="Type: Creature"/>
    </menu_literal>
  </portal>
```

Once the portal is defined, we can easily integrate it into the Position script. We're replacing the "isnpc" portal, so we can swap out all references to that portal and use "chartype" instead. The only detail that differs between menus and checkboxes is that checkboxes are automatically sized to fit the text they contain, while menus are not. This means we have to specify the width of the portal to something appropriate. We must do this before we center the portal horizontally by adding the line of code below.

```
portal[chartype].width = width
```

Our new portal should now be ready to go, with the proper selections being presented and stored internally within the field. However, our behaviors aren't quite right yet. We need to control the visibility of various portals based on the character type. The wildcard checkbox should only be visible for non-creatures, while the starting cash and XP should only be visible for PCs. This requires changing the block of lines at the start of the script to be the following.

```
 ~determine whether the wildcard option is visible based on
whether we're a creature
  portal[iswild].visible =  !hero.tagis[Hero.Creature]

 ~determine whether the starting cash is visible based on
whether we're a pc
  portal[cash].visible = hero.tagis[Hero.PC]
  portal[lblcash].visible = portal[cash].visible

 ~determine whether the starting xp is visible based on if we're a
pc
  portal[xp].visible = hero.tagis[Hero.PC]
  portal[lblxp].visible = portal[xp].visible
```

New Creature Component

We could adapt the existing "Race" component for use with creatures, but creatures behave quite differently from normal characters. As such, we should define a new component and component set for handling creatures. We'll use the id "Creature" for both, and we'll force the creature specification to be unique for every actor. This yields the following basic definitions, which we'll extend below.

```
<component
  id="Creature"
 name="Creature"
  autocompset="no">
  </component>

<compset
  id="Creature"
 forceunique="yes">
  <compref component="Creature"/>
  </compset>
```

Some creatures are wildcards, while others are not. Based on the way creatures are presented in the rulebook, the wildcard designation is fixed for each particular creature. We've already omitted the checkbox from the "Configure Hero" form, so we need a way to designate a creature as a wilcard or not within its definition.

We could easily use a field on the "Creature" component for this purpose. However, the best way to handle either-or conditions for users is to utilize a tag. If the tag is present, the condition exists, else it doesn't - clean and simple. So we'll define a new tag that can be specified on the creature and give it the "Wildcard" unique id. Since this tag will be user-defined as part of the thing definition, we'll define the tag within the "User" tag group. Any creature that should be behave as a wildcard must be assigned this tag.

Now we need to do something with the tag. The actual wildcard behavior is managed via the "acIsWild" field on the "Actor" component. So we need to translate the tag into an appropriate modification of the field. This can be accomplished via an Eval script on the "Creature" component. We simply set the field value based on the presence of the tag. The appropriate Eval script looks like the one below.

```
<eval index="1" phase="Initialize" priority="5000"><![CDATA[
  trustme
  herofield[acIsWild].value = tagis[User.Wildcard]
  ]]></eval>
```

We should also track the creature type on the actor, just like we track the race. We can achieved that by defining an identity tag on each creature and forwarding that tag to the hero. This entails the following additions.

```
<identity group="Creature"/>

<eval index="2" phase="Setup" priority="2000"><![CDATA[
  perform forward[Creature.?]
  ]]></eval>
```

We can now define creatures that don't do anything useful, but the framework is in place that we can build upon.

Selecting the Creature

The next thing we'll be tempted to do is start solving how to implement creatures internally. However, that process is going to entail some iterative evolution. So the best thing we can do next is setup a means of selecting a creature, allowing us to test our implementation along the way.

The race is selected on the "static" form at the top of the main window, so we might as well do the same for the creature type. The question becomes whether we want to use the same chooser for both race and creature, or use a separate chooser for each. Through

the use of various scripts and tag expressions, we could dynamically configure the chooser appropriately to each context. However, it's probably going to be a good bit simpler to manage a separate chooser, so that's what we'll do.

Creating our new chooser portal is rather simple. We can start by cloning the one for the race and then adapting it. We change the id, component, and text shown in the scripts. This results in the new portal shown below.

```
<portal
  id="stCreature"
  style="chsNormal"
  width="110">
  <chooser_table
    component="Creature"
    choosetemplate="LargeItem">
    <chosen><![CDATA[
      if (@ispick = 0) then
        @text = "{text ff0000}Select Creature Type"
      else
        @text = "Type: " & field[name].text
        endif
      ]]></chosen>
    <titlebar><![CDATA[
      @text = "Choose the type for your creature"
      ]]></titlebar>
    </chooser_table>
  </portal>
```

We now need to integrate the new portal into the layout, which starts with adding a new "portalref" for the chooser. Based on the nature of the character, we must show either the race or creature chooser. We also need to adjust the positioning of the "stActor" template to depend on whichever portal is being shown. The revised layout should look like the following.

```
<layout
  id="static">
  <portalref portal="stRace" taborder="20"/>
  <portalref portal="stCreature" taborder="20"/>
  <templateref template="stName" thing="heroname"
taborder="10"/>
  <templateref template="stActor" thing="actor"/>
  <!-- This script sizes and positions the layout and its child
visual elements. -->
  <position><![CDATA[
    ~determine whether the race of creature chooser is visible
    if (hero.tagis[Hero.Creature] <> 0) then
      portal[stRace].visible = 0
    else
      portal[stCreature].visible = 0
      endif

    ~position the name template on the left, with a little margin,
then render it
    ~to generate the appropriate dimensions
    template[stName].left = 10
    perform template[stName].render

    ~position the race/creature portal to the right of the name
    var x as number
    x = template[stName].right + 20
    if (portal[stRace].visible <> 0) then
      portal[stRace].left = x
      x = portal[stRace].right
    else
      portal[stCreature].left = x
      x = portal[stCreature].right
      endif

    ~position the actor template a little to the right of the
    template[stActor].left = x + 15

    ~limit the width of the actor template to the remaining space
available and
    ~then render the template
    template[stActor].width = width - template[stActor].left
    perform template[stActor].render

    ~center all visual elements vertically
    ~Note: We can't do this until after we've calculated the
heights for both
    ~      templates, which is done when we render them.
    portal[stRace].top = (height - portal[stRace].height) / 2
    portal[stCreature].top = (height - portal[stCreature].height) / 2
    template[stName].top = (height - template[stName].height) /
2
    template[stActor].top = (height - template[stActor].height) / 2
    ]]></position>

  </layout>
```

There are two basic problems at this point. The first is simple to fix. The width of the race chooser is good for races. However, there are a variety of creatures with rather long names (e.g. "Alligator/Crocodile" and "Shark, Great White"). The current width of the portal is insufficient, so we'll widen it. We can solve this by changing the "width" attribute on the portal to something like "170".

The other problem we face is if the user does something we aren't handling. What if the user creates a normal character, selects a race, and the decides to change the character to be a creature. The race portal will disappear, but the selected race will still exist on the character. That means any abilities or bonuses conferred by the race will continue to apply, although there will be no way to change it. More importantly, there is no way to select "none" for the race, so any race the user selects will persist for the life of the character. That's not very useful.

What we need is a way to ensure that the race gets deselected if the character type changes to a creature. Just in case the user goes the other direction, we also need a way to ensure that any selected creature type is discard if the character type changes away from being a creature. Fortunately, the Kit provides a convenient mechanism for accomplishing exactly this behavior

The mechanism is an Existence tag expression, which establishes requirements for the continued existence of picks. This tag expression is defined for a table or chooser and works just like a ContainerReq tag expression. Any pick that is added to a container via the portal inherits the Existence tag expression of the portal. The tag expression is then evaluated at a specified time during the evaluation cycle. If the tag expression ever fails to be satisfied, it fails its existence requirements and HL automatically deletes the pick.

We can put this to use with our two choosers. Any race added via the "stRace" chooser must be discarded if the character type becomes a creature. This results in an Existence tag expression that requires the container (i.e. the character) to not possess the "Hero.Creature" tag. We have to schedule the test to be performed before any scripts that depend on the race occur, but after the tag is assigned to the actor. So we add the following XML to the "stRace" chooser to impose the requirement.

```
<existence phase="Initialize"
priority="2000">!Hero.Creature</existence>
```

Similarly, the "stCreature" chooser needs to enforce an Existence tag expression that is just the opposite in behavior. If the character type ever ceases to be a creature, the creature type must be discarded. This results in the XML element below being added to the chooser.

```
<existence phase="Initialize"
priority="2000">Hero.Creature</existence>
```

At this point, switching between a creature and non-creature will automatically discard any existing selection corresponding to the other type.

TEST THE BASICS

We can now do some basic testing of our new changes to make sure that they're working properly. We can define a simple creature that doesn't do much, but we can verify all the fields and tags are being setup properly. We can also confirm that our choosers are behaving properly and the Existence tag expression is being handled correctly. So we'll define a simple creature like the one below.

```
<thing
  id="creSample"
  name="Sample"
  compset="Creature"
  isunique="yes"
  description="Description goes here">
  </thing>
```

After experimenting a bit, there is one problem that we can uncover. If we assign the "User.Wildcard" tag to a creature, it is given three Bennies instead of two, which means it's being treated as a normal character. Looking at the tracker we defined for Bennies, it seems our initialization logic is the problem. We assign two Bennies to NPCs and three to all other wildcards. We need to change the logic to assign two Bennies to any non-PC, which means verifying that the "Hero.PC" tag is not present on the actor. The line of code in the script that checks for an NPC must be changed to the one shown below.

```
elseif (hero.tagis[Hero.PC] = 0) then
```

CREATURE SPECIFICATION (SAVAGE)

It's now time to figure out how we're going to be handling the details of creatures. Each creature in the rulebook has die-types specified for each attribute. Creatures also have a list of skills, with the appropriate die-type for each. Derived traits are specified, incorporating any special adjustments beyond the standard derived value. Lastly, a list of gear and/or special abilities is given.

Our solution for creatures must enable all of these different facets to be readily assigned to each creature. We want a design that is both easy to use and also easy to utilize via the integrated Editor. For the latter, it's always easiest when tags are utilized, since users can simply pick from a controlled list that the Editor presents. The next easiest mechanism is simple fields, wherein the user can enter text or a value. After that, whatever works is going to be pretty much the same.

Given the list of requirements and goals above, we'll see about devising a good solution. As an added user benefit, we'll also strive to allow the user to customize the default values for each creature if he wants.

INTERNAL APPROACH

There are two different facets of traits that we'll need to handle. The first is the basic die-type for the trait, ranging from "d4" to "d12". The second is the roll bonus in situations where the creature has a trait that exceeds a "d12" rating.

These two facets are managed internally via two separate fields for each trait. As such, our first thought will likely be to do the same. We could have two separate fields that specify the starting die-type and bonus for each trait. An Eval script could be written that assigns the die-type value to the "trtBonus" field and the bonus value to the "trtRoll" field.

This would give us a workable solution, but it would be far from ideal. Since we're applying the values as bonuses, we run into a number of limitations. The most important limitation is that we are unable to allow the user to adjust the trait values. Since the attributes start at the minimum and the bonus is applied, users can increase the die-type but not decrease it. Consequently, we can't allow adjustment, since users will expect full range of adjustment to be possible. Another issue is that users would be unable to adjust the roll bonus. If the user wanted to change a trait from "d12+2" to "d12+1", there would be no way to do it, other than applying a permanent adjustment to the creature.

Giving the problem a bit more thought, a better approach would be to actually set the "trtUser" field of the trait to the proper die-type. This would make it possible for the user to then edit the die-type as he saw fit. Unfortunately, there are two liabilities with this technique. There would still be no way for the user to easily adjust any roll bonus, plus there would be nothing to prevent the user from modifying the die-type independently from the roll bonus. For example, a trait of "d12+1" could have only the die-type modified downwards one notch, which would result in a trait of "d10+1" instead of "d12".

Based on this analysis, we need a single unified value that represents both the die-type and the roll bonus. Once the value reaches a "d12" rating, increasing it one step would become "d12+1". The increase would steadily adjust the roll bonus as soon as the maximum die-type was reached. How can we do this?

The trick is to modify the behavior of the "trtUser" field when dealing with a creature. For a normal character, the "trtUser" field has a fixed range of 2-6, corresponding to the "d4" through "d12" die-types. However, we already have to handle the case of the die-type exceeding "d12" when determining what to display, so having a value of 8 will still result in a "d12" being displayed. This means we can change the maximum limit to higher than 6 and the die-type display will continue to work fine.

Armed with this knowledge, we can consider values larger than 6 to indicate progressive increases in the roll bonus. A value of 7 would translate to "d12+1", while 8 would translate to "d12+2", etc. We can write an Eval script to detect a value greater than 6 and automatically apply the excess as a roll bonus by modifying the "trtRoll" field.

Since the "trtUser" field is directly modifiable by the user via the incrementer, this approach affords the user with a simple, intuitive

means of adjusting traits. They start out at "d4", progress upwards through "d12", and then continue to increase as "d12+N". Decrementing the trait works seamlessly in the other direction.

We've got our strategy mapped out, so let's put the basic mechanisms into place. There are two changes we need to make internally to support this approach. The first is to setup a different maximum value for traits when creating a creature. We'll support up to "d12+10", so we'll set out maximum accordingly. The Eval script where the ranges are specified can be changed to the following.

```
<eval index="3" phase="Initialize" priority="3000"><![CDATA[
  ~since die types range from d4 to d12, in multiples of 2, we
have a range of
  ~2-6 for traits - we'll convert to the die type for display in a
separate script
  ~Note: Creatures treat excess as a roll bonus, so let the
maximum go to 16 (+10).
  field[trtMinimum].value = 2
  if (hero.tagis[Hero.Creature] <> 0) then
    field[trtMaximum].value = 16
  else
    field[trtMaximum].value = 6
    endif
  ]]></eval>
```

With the maximum relaxed, we can add the handling for a value that exceeds 6. We'll define a new Eval script that only does something when we have a creature with a trait that exceeds 6. If we do, then the excess is added to the "trtRoll" field and becomes a roll bonus. The new Eval script should look like the one below.

```
<eval index="5" phase="Traits" priority="8000">
  <before name="Calc trtNetRoll"/><![CDATA[
    ~for creatures with a die-type greater than 6, the excess is a
roll bonus
    if (hero.tagis[Hero.Creature] <> 0) then
      if (field[trtUser].value > 6) then
        field[trtRoll].value = field[trtUser].value – 6
        endif
      endif
  ]]></eval>
```

External Approach

We also have to figure out a convenient way to let authors specify the die-type and roll bonus values for each trait. Whatever we come up with should be readily adaptable for use within the integrated Editor, thereby allowing users to add their own custom creatures easily.

As above, the first idea is probably going to be two separate fields that can be specified. Since we're only assigning starting values, we don't have to worry about having a smooth progression. However, two separate values does make it possible for the user to inadvertently specify a roll bonus with a die-type less than "d12", such as "d8+2". So a single value would be ideal.

If we only have a single value, we then need to assess what that value will be and how the user will specify it. We currently use a value of 2-6 for the die-types "d4" through "d12". Our internal solution calls for using values of 7+ to indicate the roll bonus beyond a "d12". While all this makes perfect sense for internal use, it's going to seem like nonsense to a user with no knowledge about the inner workings of our data files. An most users don't want to understand the inner workings - they just want to knock out a quick creature for use in the upcoming game and be done with it. That means field values are a poor solution.

The best solution is to use an assortment of tags. We could define a new tag group that contains tags corresponding to each internal value we want to support. Each tag could be defined with a suitable publicly visible name that presents something intuitive to the user. For example, the tag for the value "7" would have a name of "d12+1", so the user would select "d12+1" within the Editor and the proper value of 7 would be assigned internally. We can use the "tagvalue" target reference within the scripting language to extract a value from the unique id of each tag, so we simply need to ensure that the unique id of the tags is the value we want to use internally.

An example of what the tag group should look like is provided below. We define tags that extend up to a "+10" bonus on the trait, which will be the maximum we support.

```
<group
  id="DieType"
  name="Die-Type">
  <value id="2" name="d4"/>
  <value id="3" name="d6"/>
  <value id="4" name="d8"/>
  <value id="5" name="d10"/>
  <value id="6" name="d12"/>
  <value id="7" name="d12+1"/>
  <value id="8" name="d12+2"/>
  <value id="9" name="d12+3"/>
  <value id="10" name="d12+4"/>
  <value id="11" name="d12+5"/>
  <value id="12" name="d12+6"/>
  <value id="13" name="d12+7"/>
  <value id="14" name="d12+8"/>
  <value id="15" name="d12+9"/>
  <value id="16" name="d12+10"/>
  </group>
```

At this point, we now have both our internal and external approaches mapped out. We can now begin implementing our solution.

Assigning Attributes

We'll first setup the means for specifying the attribute values to be used when defining creatures. We need to use a group of tags to specify both the die-type and roll bonus for each attribute. However, if we only define one set of tags and re-use them, we'll run into the problem of determining which tag is associated with which attribute. This means we need a separate set of tags for each attribute. Each of the tag groups is shown below, and the full set of tags defined in the previous section must be specified for each tag group.

```
<group
  id="AgiDie"
  name="Agility Die-Type">
  ...
  </group>

<group
  id="SmaDie"
  name="Smarts Die-Type">
  ...
  </group>

<group
  id="SpiDie"
  name="Spirit Die-Type">
```

```
  ...
  </group>

<group
  id="StrDie"
  name="Strength Die-Type">
  ...
  </group>

<group
  id="VigDie"
  name="Vigor Die-Type">
  ...
  </group>
```

We now have five tag groups, each with its own set of tags. Each creature can be assigned a single tag from each tag group, with each tag dictating the appropriate die-type and roll bonus for its corresponding attribute. For example, the creature below would be assigned die-types of "d4" through "d12" for the various attributes.

```
<thing
  id="creSample"
  name="Sample"
  compset="Creature"
  isunique="yes"
  description="Description goes here">
  <tag group="AgiDie" tag="2"/>
  <tag group="SmaDie" tag="3"/>
  <tag group="SpiDie" tag="4"/>
  <tag group="StrDie" tag="5"/>
  <tag group="VigDie" tag="6"/>
  </thing>
```

The final piece we need to handle is taking the values dictated by the tags and assigning them to the appropriate attributes. Our first instinct will be to create a new Eval script. In that script, we can assign the value of each tag to the "trtUser" field of the trait. Unfortunately, that won't work.

The problem is that using an Eval script will result in the value being applied to the field during every evaluation cycle. If we do that, the user can modify the attribute all he wants, but we'll always keep resetting the value back to the initial value for the creature. That's not what we had in mind.

Fields of type "user" work differently from "derived" fields. A "derived" field is reset at the start of each evaluation cycle and must be set to an appropriate value within each cycle. In contrast, a "user" field is only ever changed when the user makes a change or a script assigns a new value. Consequently, what we need to do is assign the proper values to the "user" fields when the creature is created and never touch them again. That way, we initialize them and hand control to the user thereafter.

The Kit provides a mechanism to handle exactly this type of situation. Every component can define a Creation script. This script is invoked exactly once for every pick derived from that component - at the time the pick is first created. If we assign the values within this script, they will be setup once and never modified again. The appropriate Creation script for setting up the various attributes is shown below.

```
<creation><![CDATA[
  ~assign the appropriate die-type ratings to attributes
  #traituser[attrAgi] = tagvalue[AgiDie.?]
  #traituser[attrSma] = tagvalue[SmaDie.?]
  #traituser[attrStr] = tagvalue[StrDie.?]
  #traituser[attrVig] = tagvalue[VigDie.?]
  ]]></creation>
```

Reload the data files, create a creature, and then select the sample creature above. If everything is working, each of the attributes should be initialized to the proper die-type. Modifying the various attributes up and down via the incrementer will properly adjust the attributes, allowing the user to fully control everything after the creature is first added.

The one limitation of this approach is that changing the creature will reset any adjustments made by the user. In general, that behavior is a good thing, since a new creature should always start out with the default values assigned in the rulebook.

Assigning Skills

The overall technique for assigning skills is very similar to the one used for attributes. However, there is a very important wrinkle in dealing with skills. In Savage Worlds, skills are not added to every character, so that means creatures don't begin with any skills. The technique we used for attributes assumes that the attribute is already on the character so that it can be properly adjusted.

The only way to add a skill to a character is by bootstrapping it. This means that each creature must bootstrap each of the skills that it possesses. Once we do that, though, we need a way to initialize the "trtUser" field of the skill appropriately. If we try to do anything within the Creation script of the "Creature" component, it will fail, since the script is invoked immediately upon creation, which is before any bootstrapped picks are added to the creature.

The solution is to leverage the Creation script on the skill itself. We can assign the appropriate die-type tag to the skill as part of the bootstrap process. That tag is considered part of the skill, so it can be utilized within the Creation script of the skill.

This means that we only need a single tag group for all skills. Since the proper tag will be assigned to each separate skill, no skill should have more than one such tag. So we can define a new tag group like the one below that contains all of the various tags for the different die-types.

```
<group
  id="SkillDie"
  name="Skill Die-Type">
  ...
  </group>
```

With the tag group in place, we can assign a few skills to our sample creature for testing. Each skill is bootstrapped and assigned the proper tag for the die-type it should possess. The revised sample creature below demonstrates three skills be added to the creature. These skills are then assigned die-types of "d6", "d8", and "d10".

```
<thing
  id="creSample"
  name="Sample"
  compset="Creature"
  isunique="yes"
  description="Description goes here">
  <tag group="AgiDie" tag="2"/>
  <tag group="SmaDie" tag="3"/>
  <tag group="SpiDie" tag="4"/>
  <tag group="StrDie" tag="5"/>
  <tag group="VigDie" tag="6"/>
```

```
<bootstrap thing="skFighting">
  <autotag group="SkillDie" tag="3"/>
  </bootstrap>
<bootstrap thing="skGuts">
  <autotag group="SkillDie" tag="4"/>
  </bootstrap>
<bootstrap thing="skNotice">
  <autotag group="SkillDie" tag="5"/>
  </bootstrap>
</thing>
```

The skills are automatically added to the creature and assigned their default rating of "d4". We now need to do something with the tags to initialize the die-type ratings. We'll define a Creation script for the "Skill" component. In this script, we'll first check to make sure that we have a creature and a suitable tag with which to initialize the skill. We can then use the same approach as with attributes, assigning the value of the tag as the initial value of the "trtUser" field. The Creation script below shows the logic in action.

```
<creation><![CDATA[
  ~if this is a creature with a skill rating, assign the die-type
rating to the skill
  if (hero.tagis[Hero.Creature] + tagis[SkillDie.?] >= 2) then
    field[trtUser].value = tagvalue[SkillDie.?]
    endif
  ]]></creation>
```

At this point, skills should be working smoothly for creatures.

Assigning Derived Traits

Derived traits work quite differently from attributes and skills for normal characters. That's because they are 100% derived from facets of the character. For creatures, though, we need to handle them differently. Various creatures have values assigned for certain derived traits that are not purely derived. The most common trait is "Pace", but other traits vary as well.

The problem is that derived traits are always calculated in the end. We need to handle them in a way that will smoothly integrate with their calculated nature. Unfortunately, there is no clean way of doing that by having the derived trait value be specified explicitly for a creature. The only way to reasonably accomplish it is to utilize an adjustment that gets incorporated into the calculated final value.

This means that we need to define four new fields for the "Creature" component. We need one field apiece for the four derived traits, where each field specifies an adjustment from the standard calculated value. If a field is not specified, then no extra adjustment will be applied for that trait.

From a usage standpoint, the problem with this approach centers on Pace. Virtually all creatures have a custom Pace, so we should really assume that Pace has a starting value of zero instead of the current six. That way, the actual Pace can be specified instead of an adjustment. Since the other derived traits rarely differ from the calculated value, requiring an adjustment to the calculated value is perfectly reasonable.

In order to specify the Pace as an explicit value, we need to modify the trait itself. The "trPace" trait currently assumes a default value of six for the "trtBonus" field. We need to override that behavior when we have a creature. So we'll define the following Eval script on the trait to apply the special handling.

```
<eval index="2" phase="Initialize" priority="3000"><![CDATA[
  if (hero.tagis[Hero.Creature] <> 0) then
    field[trtBonus].value = 0
    endif
  ]]></eval>
```

With the change in place, we can shift back to the fields on the "Creature" component. We'll end up with four fields, where the Pace indicates the actual starting value to use and the others indicate an adjustment. Our new fields should looks like the following.

```
<field
  id="crePace"
  name="Base Pace"
  type="static">
  </field>

<field
  id="creParry"
  name="Parry Adjustment"
  type="static">
  </field>

<field
  id="creTough"
  name="Toughness Adjustment"
  type="static">
  </field>

<field
  id="creCharis"
  name="Charisma Adjustment"
  type="static">
  </field>
```

We now need to communicate the adjustment value for each derived trait from the creature. Each derived trait needs to know what the particular adjustment value is. We'll define a new field on the "Derived" component for this purpose, which should look like below.

```
<field
  id="trtCreatur"
  name="Creature Start"
  type="derived">
  </field>
```

Each creature can tell its derived traits the value to be used via an Eval script on the "Creature" component. This script must be invoked early in the evaluation cycle, but it is otherwise very simple. The result script is shown below.

```
<eval index="3" phase="Setup" priority="5000"><![CDATA[
  ~setup the proper starting values for each trait
  hero.child[trPace].field[trtCreatur].value = field[crePace].value
  hero.child[trParry].field[trtCreatur].value = field[creParry].value
  hero.child[trTough].field[trtCreatur].value =
field[creTough].value
  hero.child[trCharisma].field[trtCreatur].value =
field[creCharis].value
  ]]></eval>
```

At this point, each derived trait should know the adjustment value specified by any selected creature. Integrating that value into the final calculation for the derived trait is relatively simple. All we need to do is revise the Eval script that calculates the value of "trtFinal"

to add in the adjustment for creatures. We can achieve this by adding the lines of code below to the Eval script.

```
~if this is a creature, we need to add the user value as a custom
adjustment
 if (hero.tagis[Hero.Creature] <> 0) then
  field[trtFinal].value += field[trtCreatur].value
  endif
```

That's all there is to it. We're now ready to give our changes a try. When we select a creature from the chooser, the appropriate values should be applied as adjustments to the final derived trait values. Let's put our changes to the test with an actual creature that applies adjustments.

We'll modify our sample creature for this purpose. We can assign a few field values on the creature to apply our adjustments. We'll assign a "Pace" of "5" and a "Parry" adjustment of "1". To do this, we'll add the following XML elements to the creature's definition.

```
<fieldval field="crePace" value="5"/>
 <fieldval field="creParry" value="1"/>
```

Reload the data files and create a creature. Take a look at the default values shown for the derived traits. From the chooser, select our test creature. The "Pace" value should immediately change to "5", and the "Parry" value should change from "2" to "3". Our derived traits are working smoothly now.

Assigning Gear

The assignment of standard gear to creatures is incredibly simple. All that's needed is to bootstrap the desired gear onto the creature. If the gear is a weapon or armor, then you may also want to have the gear start out in the "equipped" state, which can be accomplished by assigning the "Equipment.StartEquip" tag to the gear. For example, a "Dagger" can be automatically assigned to a creature and pre-equipped via the XML element below.

```
<bootstrap thing="wpDagger">
  <autotag group="Equipment" tag="StartEquip"/>
  </bootstrap>
```

Many creatures also possess a natural attack of some sort and/or natural armor. You can add these as weapons in the same way as other gear. However, you're much better off using the customizable abilities that we defined earlier for this purpose.

Assigning Abilities

The assignment of abilities to a creature is comparable to assigning gear. Once the ability is defined, a simple "bootstrap" element is all that's needed. The definition of abilities has already been outlined previously. For example, the "Rollover" special ability of an Alligator can be defined as shown below and assigned to a creature via the bootstrap beneath.

```
<thing
  id="abRollover"
  name="Rollover"
  compset="RaceAbil"
  isunique="yes"
  description="Description goes here">
  </thing>
```

```
<bootstrap thing="abRollover"/>
```

If a creature has a natural weapon, it's easiest to use the customizable weapon we defined earlier. No new ability needs to be defined, as the existing "Unarmed Attack" weapon can be tailored to the particulars necessary for the creature. For example, the "Bite" attack of the Alligator can be assigned to the creature via the following.

```
<bootstrap thing="abWeapon">
  <assignval field="livename" value="Bite"/>
  <assignval field="abilText" value="+d6"/>
  </bootstrap>
```

A similar mechanism was defined for use with natural armor and can be readily employed. All that's needed is the defensive bonus conferred. For example, the thick skins of the Alligator confer a "+2" armor bonus, which can be assigned via the following.

```
<bootstrap thing="abArmor">
  <assignval field="abilValue" value="2"/>
  </bootstrap>
```

Creature Refinement (Savage)

We've now got the ability to define and customize creatures in place. However, there are still a fair number of tasks we need to complete before creatures are fully handled. The sections below address these remaining issues.

Organizing Abilities Between all the races and creatures, we've got a long list of special abilities. These abilities seem to break down into five different classifications.

- Common racial abilities that can be shared by multiple races (e.g. Tough, Agile)
- Race-specific abilities that are unique to a single race (e.g. Avion Flight, Saurian Senses)
- Common creature abilities that can be shared by multiple creature (e.g. Size, Infravision)
- Creature-specific abilities that are unique to a single creature (e.g. Rollover, Bear Hug)
- Generic abilities that are shared by both races and creatures (e.g. Natural Weapons, Natural Armor)

To keep everything organized, we'll partition these abilities across three separate data files:

- One file will contain all the races and race-specific abilities ("thing_races.dat")
- Another file will contain all the creatures and creature-specific abilities ("thing_creatures.dat")
- The third file will contain only re-usable special abilities for both races and/or creatures ("thing_abilities.dat")

We need a way to readily identify common creature abilities as such. For this, we'll define a new "User.Creature" tag, which we can then assign to all abilities that can be used freely by any creature. These abilities will consist of the two generic abilities we created for natural weapons and armor, as well as all of the common creature abilities spelled out in the rulebook.

Defining Abilities

We can now focus on getting all of the common creature abilities properly defined. The core rulebook outlines a couple dozen of

these abilities. We should get them all into place before we start defining lots of creatures that depend on them.

We'll pick a few of these abilities to implement as examples here. We'll start with the "Size" ability, which specifies a non-standard size for the creature. The ability requires the specification of a size bonus (or penalty). We'll utilize the customizable ability mechanism and define the size bonus via the "abilValue" field. This bonus is both applied to the "Toughness" trait as an adjustment and incorporated into the name for display. This results in the ability definition below, which is followed by an example of its use.

```
<thing
  id="abSize"
  name="Size"
  compset="RaceAbil"
  isunique="yes"
  description="Description goes here">
  <tag group="User" tag="Creature"/>
  <!-- Apply the size bonus to the Toughness trait -->
  <eval index="1" phase="PreTraits" priority="5000">
    <before name="Calc trtFinal"/><![CDATA[
    perform #traitadjust[trTough,+,field[abilValue].value,"Size"]
    field[livename].text = field[name].text & " " &
signed(field[abilValue].value)
    ]]></eval>
  </thing>

<bootstrap thing="abSize">
  <assignval field="abilValue" value="2"/>
  </bootstrap>
```

Our next example is the "Immunity" ability. This ability requires that specification of the nature of the immunity. The customizable ability mechanism can again be used, only this time we'll specify the specific immunity via the "abilText" field. The immunity can then be integrated into the name for the display. This yields the ability definition below, followed by an example of its use.

```
<thing
  id="abImmunity"
  name="Immunity"
  compset="RaceAbil"
  description="Description goes here">
  <tag group="User" tag="Creature"/>
  <!-- Append the immunity to the name -->
  <eval index="1" phase="Traits" priority="10000"><![CDATA[
    field[livename].text = field[thingname].text & ": " &
field[abilText].text
    ]]></eval>
  </thing>

<bootstrap thing="abImmunity">
  <assignval field="abilText" value="Fire"/>
  </bootstrap>
```

Some abilities automatically confer other abilities. For example, the "Elemental" ability confers the "Fearless" ability. This is easily achieved by simply bootstrapping the conferred ability, as shown in the definition below.

```
<thing
  id="abElement"
  name="Elemental"
  compset="RaceAbil"
  isunique="yes"
  description="Description goes here">
  <tag group="User" tag="Creature"/>
  <bootstrap thing="abFearless"/>
  </thing>
```

You should now be able to implement all the various abilities utilized in the rulebook. Alternately, you can simply look at the completed Savage Worlds data files provided with HL.

New Abilities Tab

We eliminated the "Edges" tab with the assumption that we would replace it with a suitable alternative. We'll take care of that now. What we need is a tab where all the abilities assigned to the creature can be shown to the user. We only need one table, so we can clone the "Skills" tab and adapt to our needs.

Start by copying the "tab_skills.dat" file to "tab_abilities.dat", then open the new file for modification. We need to define a new table portal. Since all the abilities are static for a creature, we simply need a fixed table. All we need to show is the name of each ability, so we can use the "SimpleItem" template to show the abilities. This results in the portal definition shown below.

```
<portal
  id="abAbility"
  style="tblNormal">
  <table_fixed
    component="RaceAbil"
    showtemplate="SimpleItem">
    <headertitle><![CDATA[
      @text = "Monstrous Abilities"
    ]]></headertitle>
  </table_fixed>
  </portal>
```

We can then adapt the layout to show our new portal and have its own identity. The new layout should look like the following.

```
<layout
  id="abilities">
  <portalref portal="abAbility" taborder="10"/>
  <position><![CDATA[
    ~position and size the table to span the full layout; it will only
use the
    ~vertical space that it actually needs
    perform portal[abAbility].autoplace
    ]]></position>
  </layout>
```

The final step is to revise the panel definition. We want this panel to appear in place of the "Edges" panel, so we'll assign it the same "order" value as that panel. We also want this panel to only appear for creatures, so we'll use a Live tag expression that requires the presence of the "Hero.Creature" tag. After we modify the name and other references appropriately, we end up with the panel below.

```
<panel
  id="abilities"
  name="Abilities"
  marginhorz="5"
  marginvert="5"
  order="130">
  <live>Hero.Creature</live>
  <layoutref layout="abilities"/>
  <position><![CDATA[
    ]]></position>
  </panel>
```

We're now ready to give it a try. Reload the files and create a creature. The new tab should appear with a list of the various abilities assigned to the creature.

Defining Complete Creatures

All the pieces are in place to allow you to define any of the creatures in the core rulebook. We'll take this opportunity to define a few of them here as examples.

The first example we'll use is the "Alligator/Crocodile" creature. This creature has a non-standard "Pace" on the ground, as well as a special "Pace" in the water. It utilizes both a natural weapon and natural armor. Plus it has a creature-specific ability. The full definition of the creature and its new special ability is shown below.

```
<thing
  id="creAlligat"
  name="Alligator/Crocodile"
  compset="Creature"
  isunique="yes"
  description="Description goes here">
  <fieldval field="crePace" value="3"/>
  <tag group="AgiDie" tag="2"/>
  <tag group="SmaDie" tag="2"/>
  <tag group="SpiDie" tag="3"/>
  <tag group="StrDie" tag="5"/>
  <tag group="VigDie" tag="5"/>
  <bootstrap thing="skFighting">
   <autotag group="SkillDie" tag="4"/>
   </bootstrap>
  <bootstrap thing="skGuts">
   <autotag group="SkillDie" tag="3"/>
   </bootstrap>
  <bootstrap thing="skNotice">
   <autotag group="SkillDie" tag="3"/>
   </bootstrap>
  <bootstrap thing="skSwimming">
   <autotag group="SkillDie" tag="4"/>
   </bootstrap>
  <bootstrap thing="abArmor">
   <assignval field="abilValue" value="2"/>
   </bootstrap>
  <bootstrap thing="abWeapon">
   <assignval field="livename" value="Bite"/>
   <assignval field="abilText" value="+d6"/>
   </bootstrap>
  <bootstrap thing="abAquatic">
   <assignval field="abilValue" value="5"/>
   </bootstrap>
  <bootstrap thing="abRollover"/>
  </thing>

<thing
  id="abRollover"
  name="Rollover"
  compset="RaceAbil"
  isunique="yes"
  description="Description goes here">
  </thing>
```

For our next example, we'll use the "Large Bear" entry. This creature introduces the use of a Strength rating above "d12". It also utilizes a natural weapon and the "Size" ability, as well as its own custom ability. The full definition of the creature is shown below.

```
<thing
  id="creBearLg"
  name="Bear, Large"
  compset="Creature"
  isunique="yes"
  description="Description goes here">
  <tag group="AgiDie" tag="3"/>
  <tag group="SmaDie" tag="3"/>
  <tag group="SpiDie" tag="4"/>
  <tag group="StrDie" tag="10"/>
  <tag group="VigDie" tag="6"/>
  <bootstrap thing="skFighting">
   <autotag group="SkillDie" tag="4"/>
   </bootstrap>
  <bootstrap thing="skGuts">
   <autotag group="SkillDie" tag="5"/>
   </bootstrap>
  <bootstrap thing="skNotice">
   <autotag group="SkillDie" tag="4"/>
   </bootstrap>
  <bootstrap thing="skSwimming">
   <autotag group="SkillDie" tag="3"/>
   </bootstrap>
  <bootstrap thing="abWeapon">
   <assignval field="livename" value="Claw"/>
   <assignval field="abilText" value="+d6"/>
   </bootstrap>
  <bootstrap thing="abSize">
   <assignval field="abilValue" value="2"/>
   </bootstrap>
  <bootstrap thing="abBearHug"/>
  </thing>

<thing
  id="abBearHug"
  name="Bear Hug"
  compset="RaceAbil"
  isunique="yes"
  description="Description goes here">
  </thing>
```

For our final example, we'll implement a creature that's got some interesting twists. The "Lich" is a wildcard, has skills that exceed a "d12" rating, and has a skill that requires a domain be specified. It also has gear in the form of magical armor and gains arcane powers with extra bonuses to spells and power points. The full implementation is shown below (the "Undead" ability is assumed already defined as one of the common abilities shared by creatures).

```
<thing
  id="creLich"
  name="Lich"
  compset="Creature"
  isunique="yes"
  description="Description goes here">
  <fieldval field="crePace" value="6"/>
  <tag group="User" tag="Wildcard"/>
  <tag group="AgiDie" tag="3"/>
  <tag group="SmaDie" tag="8"/>
  <tag group="SpiDie" tag="5"/>
  <tag group="StrDie" tag="5"/>
  <tag group="VigDie" tag="5"/>
  <bootstrap thing="skFighting">
   <autotag group="SkillDie" tag="4"/>
   </bootstrap>
  <bootstrap thing="skGuts">
   <autotag group="SkillDie" tag="6"/>
   </bootstrap>
  <bootstrap thing="skIntimid">
   <autotag group="SkillDie" tag="6"/>
   </bootstrap>
  <bootstrap thing="skKnow">
   <autotag group="SkillDie" tag="8"/>
   <assignval field="domDomain" value="Occult"/>
   </bootstrap>
  <bootstrap thing="skNotice">
   <autotag group="SkillDie" tag="5"/>
   </bootstrap>
```

```
    <bootstrap thing="skSpellcst">
      <autotag group="SkillDie" tag="6"/>
      </bootstrap>
    <bootstrap thing="abDeathTch"/>
    <bootstrap thing="abSpellLch"/>
    <bootstrap thing="abUndead"/>
    <bootstrap thing="abZombie"/>
    <bootstrap thing="armMagLch">
      <autotag group="Equipment" tag="StartEquip"/>
      </bootstrap>
    </thing>

  <thing
    id="armMagLch"
    name="Magical Armor +6"
    compset="Armor"
    description="Description goes here"
    isunique="yes">
    <fieldval field="defDefense" value="6"/>
    <tag group="Equipment" tag="StartEquip"/>
    <tag group="ArmorLoc" tag="Torso"/>
    <tag group="ArmorLoc" tag="Arms"/>
    <tag group="ArmorLoc" tag="Legs"/>
    <tag group="ArmorLoc" tag="Head"/>
    </thing>

  <thing
    id="abDeathTch"
    name="Death Touch"
    compset="RaceAbil"
    isunique="yes"
    description="Description goes here">
    </thing>

  <thing
    id="abSpellLch"
    name="Spells"
    compset="RaceAbil"
    isunique="yes"
    description="Description goes here">
    <bootstrap thing="edgArcMag"/>
    <eval index="1" phase="PreTraits" priority="5000"><![CDATA[
      ~confer an extra 40 power points and an extra 7 spells
  (50/10 total)
      #resmax[resPowers] += 7
      #trkmax[trkPower] += 40
      ]]></eval>
    </thing>

  <thing
    id="abZombie"
    name="Zombie"
    compset="RaceAbil"
    isunique="yes"
    description="Description goes here">
    </thing>
```

Showing Edges

A number of creatures are assigned edges. Unfortunately, those edges aren't appearing on the new "Abilities" tab that we created. We need to show them appropriately so that the user clearly sees them. That requires us to expand the "Abilities" to include a second table for the edges. The good news is that we can simply re-use the existing table of edges within a second layout.

We'll replace the simplistic "abilities" layout with something that intelligently handles the two tables. We have to handle the case where both tables are too tall to fit within the available vertical space. Since edges will be generally uncommon for creatures, and their number will be few, we'll put them at the bottom, with creature abilities at the top. We'll reserve space for a few edges, then dedicate the rest of the height to abilities. If we don't need all the space for abilities, the edges can expand to fill whatever room remains. Putting all this together yields the revised layout shown below.

```
 <layout
    id="abilities">
    <portalref portal="abAbility" taborder="10"/>
    <portalref portal="edEdges" taborder="20"/>

   <!-- This script sizes and positions the layout and its child
visual elements. -->
    <position><![CDATA[
      ~size all tables to span the full layout width
      portal[abAbility].width = width
      portal[edEdges].width = width

     ~reserve space for at least three rows of edges (if we need
them)
      portal[edEdges].maxrows = 3

     ~position the edges table at the bottom
      portal[edEdges].top = height - portal[edEdges].height

     ~assign the remaining space to the abilities table
      portal[abAbility].height = portal[edEdges].top - 10 -
portal[abAbility].top

     ~position the edge table beneath abilities and let it expand to
fill space
      portal[edEdges].top = portal[abAbility].bottom + 10
      portal[edEdges].height = height - portal[edEdges].top
      ]]></position>

   </layout>
```

Cleanup the Interface

Due to our changes in how creatures are handled, there are a number of facets of the interface that need some work. We'll start with the issues on the "Basics" tab.

The first thing we can't miss on the "Basics" tab is the display of XP information and the list of creation resources. Creatures don't track any XP and they don't observe any special creation rules regarding point allocations. Consequently, both of these sections of information and any associated separators need to be hidden for creatures. This can be handled easily by adding the block of code below to the Position script for the "basics" layout.

```
~if we're creating a creature, don't show XP or creation details
 if (hero.tagis[Hero.Creature] <> 0) then
   portal[baRank].visible = 0
   portal[baCreation].visible = 0
   portal[separator1].visible = 0
   portal[separator2].visible = 0
   endif
```

There is also the matter of the header above the table of attributes. The header shows the number of remaining points that can be spent, which is meaningless for creatures. We need to revise the header to simply show the text "Attributes" when the user is creating a creature. This is accomplished by changing the code for the HeaderTitle script to that shown below.

```
@text = "Attributes"
 if (hero.tagis[Hero.Creature] = 0) then
   @text &= "  -  " & hero.child[resAttrib].field[resSummary].text
   endif
```

We can now proceed to the "Skills" tab, where we'll find a few additional issues to address. For example, the header above the list of skills suffers the same problem that the one above the attributes did on the previous tab. The solution is the same, replacing the HeaderTitle script with the new code below.

```
@text = "Skills"
 if (hero.tagis[Hero.Creature] = 0) then
  @text &= "  -  " & hero.child[resSkill].field[resSummary].text
  endif
```

If we click on the "add" item at the bottom of the table of skills, we'll see a similar problem again. The title shown at the top of the selection form shows same extra information. We need to revise the Titlebar script in the same way, as shown below.

```
@text = "Add a Skill"
 if (hero.tagis[Hero.Creature] = 0) then
  @text &= "  -  " & hero.child[resSkill].field[resSummary].text
  endif
```

The final issue on the "Skills" tab is that the "add" item at the bottom of the table is using the highlight information for a normal character. This information is controlled via the "resAddItem" field of the resource. We already disabled the special formatting for NPCs, and we need to change that logic to apply to any non-PC character (i.e. including creatures). This is done by changing the one line of code in the Finalize script to the following.

```
if (tagis[Helper.NPCImpact] +  !hero.tagis[Hero.PC] >= 2) then
```

This exact same problem repeats itself one more time on the "Abilities" tab. The table of edges needs to have both the Titlebar and HeaderTitle scripts revised to only show the summary information for non-creatures. The two new scripts are shown below, respectively.

```
@text = "Add an Edge"
if (hero.tagis[Hero.Creature] = 0) then
  @text &= "  -  " & hero.child[resEdge].field[resSummary].text
  endif
```

```
@text = "Edges"
 if (hero.tagis[Hero.Creature] = 0) then
  @text &= "  -  " & hero.child[resEdge].field[resSummary].text
  endif
```

VALIDATION RULES

The first thing we'll notice regarding the validation report is that various resources will sometimes report that they have been overspent. These are the same resources that we disabled for NPC creation, and they must also be disabled for creatures. This can be solved by modifying the Eval Rule on the "Resource" component to consider the rule as valid for any non-PC. The new line of code required is shown below.

```
if (tagis[Helper.NPCImpact] +  !hero.tagis[Hero.PC] >= 2) then
```

If the user elects to construct a creature, the specific creature type needs to be specified. We need a validation rule to verify this is done. We already a thing defined with a validation rule to verify a race is selected. We can easily clone that thing and adapt it for use with the creature type chooser, resulting in the thing shown below.

```
<thing
  id="valCreatur"
  name="Creature Type"
  compset="Simple">
  <tag group="Helper" tag="Bootstrap"/>

 <evalrule index="1" phase="Validate" priority="8000"
message="Must be selected"><![CDATA[
    ~if we have a creature type selected, we're good
    if (hero.tagis[Creature.?] <> 0) then
      @valid = 1
      done
      endif
    ]]></evalrule>

 </thing>
```

Unfortunately, we have a small problem still. If we create a normal character, we always get a validation error about the creature type. If we create a creature, we always get a validation error about the race. Each of these validation rules needs to be applied only when the appropriate character type is in use. One solution would be to utilize an appropriate ContainerReq on each of the validation things. An easier solution is to test whether we have a creature within the Eval Rule script. The revised script code for the "valRace" and "valCreatur" things is shown below.

```
~if we have a race selected or we are a creature, we're good
 if (hero.tagis[Race.?] + hero.tagis[Hero.Creature] <> 0) then
  @valid = 1
  done
  endif

~if we have a creature type selected or we're not a creature,
we're good
 if (hero.tagis[Creature.?] +  !hero.tagis[Hero.Creature] <> 0)
then
  @valid = 1
  done
  endif
```

CREATURE TYPE IN OUTPUT

The final detail we haven't addressed yet for creatures is output. Both character sheet and statblock output assume that every character will be assigned a race. We need to revise those mechanisms to handle the presence of a creature type instead.

Doing a scan through the data files, there are actually six different places where the race is being output. Each of these will need to be revised to accommodate creatures as well. This is a perfect opportunity for us to add a new procedure that can be shared by all these places. Unfortunately, we need different formatting in different situations, so a procedure will not simplify matters for us.

However, there is another solution we could use. If we add two new fields, we can setup those fields to contain the appropriate prefix and name to be used (e.g. "Race" and "Human"). For creatures, we can set the fields to use the proper info for the creature (e.g. "Creature" and "Goblin"). Once this is done, all five places can

simply retrieve the fields and not have to do any special handling for creatures.

Our two fields must be added to the "Actor" component. We'll call them "acRacePref" and "acRaceName", and we'll need to define an Eval script on the component to synthesize them properly. In case we decide to use short names anywhere for races in the future, we need to schedule our script after Render/100, but we otherwise want to have our names in place very early in the Render phase. This yields the following field definitions and script below.

```
<field
  id="acRacePref"
  name="Race Prefix"
  type="derived"
  maxlength="10">
  </field>

<field
  id="acRaceName"
  name="Race Name"
  type="derived"
  maxlength="50">
  </field>
```

```
<eval index="9" phase="Render" priority="1000"><![CDATA[
  if (hero.tagis[Hero.Creature] <> 0) then
    field[acRacePref].text = "Creature"
    field[acRaceName].text =
hero.firstchild["Creature.?"].field[name].text
  else
    field[acRacePref].text = "Race"
    field[acRaceName].text =
hero.firstchild["Race.?"].field[name].text
    endif
  if (field[acRaceName].isempty <> 0) then
    field[acRaceName].text = "-none-"
    endif
  ]]></eval>
```

Now that the fields are being setup properly, we can go through and change all references to the race to utilize the new fields. We'll start with the statblock output. The Synthesize script assumes the use of a race, and switching it to the new fields results in the following revised block of script code below.

```
~output any race or creature type
 append @boldon & herofield[acRacePref].text & ": " & @boldoff
 append herofield[acRaceName].text & @newline
```

Switching our focus over to the character sheet, the same situation exists. The "oHeroInfo" portal assumes we always use a race on the first sheet, and the "details" portal for allies makes the same assumption. We'll revise the pertinent code for each Label script to look like the following.

```
~start with the character's race
 @text &= "{size 36}" & herofield[acRacePref].text & ": " &
herofield[acRaceName].text
```

```
~output any race
 @text &= "{b}" & herofield[acRacePref].text & ":{/b} " &
herofield[acRaceName].text
```

We're halfway completed. Next up is the Eval script on the "Actor" component that synthesizes the "acRecap" field for display on the "Allies" tab. The code for outputting the race gets changed to the following.

```
~output any race
 recap &= field[acRaceName].text & ", "
```

The race is included in the text synthesized by the LeadSummary script within the definition file. We can quickly replace the code that determines the race with a reference to the new field, as shown below.

```
@text &= herofield[acRaceName].text
```

Last on our list is the procedure that synthesizes the name for the display on the Dashboard and within the Tactical Console. The "DshBasics" procedure has its race handling code changed to what's shown below.

```
~output our race
 final = herofield[acRacePref].text & ": {b}" &
herofield[acRaceName].text & "{/b}{br}{br}"
```

We can now reload our data files and verify that creatures are being handled properly within output everywhere.

CREATURE ABILITIES IN OUTPUT

A number of creatures possess an edge or two, in addition to the various creature abilities that are normally assigned. This results in two separate tables being output on the character sheet. If a hindrance is also added, then we have three tables, and none of them contain more than a few entries. It would be optimal to consolidate all of these tables into a single table for creatures.

Converting the character sheet to only use a single abilities table is pretty easy. We'll start by defining a new table portal that shows all of the abilities. It's identical to the table of racial abilities, except that we specify the general "Ability" component to encompass all the different types of abilities. The new portal is shown below.

```
<portal
  id="oCreature"
 style="outNormal">
  <output_table
    component="Ability"
    showtemplate="oAbilPick">
    <list>!Print.NoPrint</list>
    <headertitle><![CDATA[
      @text = "Creature Abilities"
     ]]></headertitle>
   </output_table>
  </portal>
```

With the portal defined, we can integrate it into the layouts. We need to integrate it into both the left and right side layouts, since it's conceivable (albeit very unlikely) that the list of abilities will extend past the bottom of the page on the left. The process is the same for both layouts. We add a new "portalref" element, then we determine the visibility of the various table portals based on the presence of the "Hero.Creature" tag, and then we add an "autoplace" statement

to position the new portal. The revised Position script for the "oLeftSide" layout is shown below.

```
~hide and show appropriate portals for creatures and non-creatures
if (hero.tagis[Hero.Creature] = 0) then
  portal[oCreature].visible = 0
else
  portal[oAbility].visible = 0
  portal[oHindrance].visible = 0
  portal[oEdge].visible = 0
  endif

~position the hero name at the top with the hero details beneath the name
perform portal[oHeroName].autoplace[0]
perform portal[oHeroInfo].autoplace[15]

~position the tables next
perform portal[oAttribute].autoplace
perform portal[oDerived].autoplace
perform portal[oSkills].autoplace
perform portal[oSkillsDom].autoplace[0]
perform portal[oAbility].autoplace
perform portal[oHindrance].autoplace
perform portal[oEdge].autoplace
perform portal[oCreature].autoplace

~our layout height is the extent of the elements within
height = autotop
```

The "oRightSide" layout is handled the exact same way, resulting in the following revised Position script.

```
~hide and show appropriate portals for creatures and non-creatures
if (hero.tagis[Hero.Creature] = 0) then
  portal[oCreature].visible = 0
else
  portal[oEdge].visible = 0
  endif

~position the various tables appropriately
perform portal[oEdge].autoplace
perform portal[oCreature].autoplace
perform portal[oDrawback].autoplace
perform portal[oPower].autoplace

~our layout height is the extent of the elements within
height = autotop
```

We can now print out a creature like the Orc and see a single table of creature abilities instead of two tiny tables of edges and racial abilities.

Bonuses from Traits Over d12

When we added the automatic spillover of a trait die larger than "d12", we broke something. If the trait die is larger than "d12", we translate that overage into the trait roll. However, that overage is not being properly factored into the calculation of the Parry and Toughness traits.

The problem stems from an issue with timing. The Parry and Toughness traits calculate the influence of Fighting and Vigor at a timing of Traits/4000. Meanwhile, the conversion of excess trait dice to the net roll doesn't happen until Traits/8000. The Eval script that does the translation only needs to be performed after the bounding of the "trtUser" field, which occurs at Traits/1000, so we can re- schedule the script to anytime after that, as long as it also occurs before Traits/4000. We'll move it to Traits/2000.

As a safeguard against future timing issues involving these scripts, we'll also add some timing dependency relationships. We'll name each of the scripts for the derived traits at Traits/4000 with a common name, such as "Calc Derived Bonus". We can then add a "before" dependency on this name to the script we changed. We'll also add an "after" dependency to this script to ensure it gets scheduled after the bounding of "trtUser". The revised framework for the script should now look like below.

```
<eval index="5" phase="Traits" priority="2000">
  <before name="Calc trtNetRoll"/>
  <after name="Bound trtUser"/><![CDATA[
  ~script code goes here
  ]]></eval>
```

If we reload and test our data files with a creature, the roll bonus is now being factored into the trait bonus. Unfortunately, the bonus is coming out wrong. Taking a look at the scripts that calculate the bonus, it seems there is an assumption that the final trait value will never exceed six. If we eliminate that assumption, a quick change to the scripts gets things working properly. The one line that needs to be changed is shown below for the Toughness calculation. An equivalent change must be made for the Parry calculation.

```
bonus = 6 + round(hero.child[attrVig].field[trtRoll].value / 2,0,-1)
```

Creatures With Zero Pace

When we added all of the creatures, there was a situation that we could specify properly, but that is being honored by our data files. The problem is when we have a creature with a Pace of zero. Since the Pace for normal characters can never drop below one, we setup appropriate bounding directly on the "trPace" trait. So even if we specify a creature as having a Pace of zero, it gets bumped back up to one.

We can solve this by modifying the Eval script that does the bounding. If the actor has the "Hero.Creature" tag, we can assume that the actor is exempt from the bounding rule. The revised Eval script is shown below.

```
if (field[trtFinal].value <= 0) then
  if (hero.tagis[Hero.Creature] = 0) then
    field[trtFinal].value = 1
    endif
  endif
```

Ignoring Rank Requirements

A number of creatures are assigned edges that have a variety of requirements. Most of those requirements are a minimum rank. Since creatures don't have any XP, they have no rank, so they fail any tests on rank. We can silence these validation errors by pretending we have plenty of XP. Since we don't use XP anywhere for creatures, this is a safe tactic.

The easiest way to accomplish this is by inserting special handling when the total XP is calculated. That's done within the "resXP" resource thing in "thing_miscellaneous.dat". All we need to do is revise the Eval script to look like the one shown below.

```
 ~if this is a creature, give ourselves a bunch of XP to reach
legendary rank
  ~Note: This silences various validation rules for edges and the
like.
  if (hero.tagis[Hero.Creature] <> 0) then
    #resmax[resXP] += 100

 ~otherwise, our xp total is our starting xp plus accrued xp via
journal entries
  else
    field[resMax].value = herofield[acStartXP].value +
hero.usagepool[TotalXP].value
    endif
```

CREATURE CUSTOMIZATION (SAVAGE)

In the previous sections, we managed to get creatures working smoothly. However, there was one area where we definitely could have provided a bit more flexibility. We allowed the user to customize attributes and skills from their starting points, but we did not do so for derived traits and abilities.

What if the GM wants a creature that's a little bit faster or tougher than normal? The answer is that we should allow him to adjust any of the creature's derived traits in whatever way suits his needs.

What if the GM wants to create an undead version of a normal creature? The simple solution would be to add the "Undead" ability to that creature, which will automatically add the Toughness adjustment and other notes for reference.

The focus of the sections below is on adding that flexibility to our data files.

CUSTOMIZING DERIVED TRAITS

At this point, we have the basic framework in place for applying custom adjustments to derived traits for creatures. What we need to add now is the ability for the user to further tailor the derived trait values for a creature. Attributes and skills utilize an incrementer that allows the user to modify the values. If we use the same basic approach for derived traits, then the user can adjust the values freely.

LEVERAGING THE "TRTUSER" FIELD

This approach would entail assigning the adjustment value to the "trtUser" field for each derived trait. But that field is not used for those traits. We're going to need to change that behavior and use the "trtUser" field for derived traits on creatures.

Derived traits calculate their final value via an Eval script on the "Derived" component. Within that script, the "trtUser" field is never used. It would be easy to use the "trtUser" value instead of "trtCreatur" within the calculation of "trtFinal". If we properly initialize the "trtUser" field values to the adjustments via the Creation script of the "Creature" component, then we can expose the "trtUser" field via an incrementer and allow the user to modify it.

Before we go down this path, though, we should think through the implications and how it's all going to work. The starting adjustments will be loaded into the "trtUser" field, which will be modified by the user. We'll want to bound that field so that it can be adjusted within a reasonable range of the starting value (e.g. -3 to +3). This will allow the GM to tailor a creature appropriately without going completely wild. We'll then add the user value into the final trait calculation. This all seems perfectly sound.

Unfortunately, there's a critical liability in this approach. All of the current mechanisms we've implemented are built around the "trtUser" field being used with attributes and skills. The field will need to be handled in many different ways for derived traits. Our bounding limits will be different, the initial value will have to change, and how the field is utilized in various places will need to be revised. The one field will need to be used in two very different ways for different traits. That's going to make implementing our changes more complicated and will make maintaining our data files in the future much more difficult.

On top of this, there is a subtle timing issue that comes into play with the bounding of the user value. Since the "trtUser" field is also used for normal (i.e. non-derived) traits, the bounding logic needs to be changed for our derived traits. That can be handled, but the bounding must be performed before the initial calculation of "trtFinal" within the "Traits" component. For our purposes, we need to access the net values calculated for our derived traits in order to do proper bounding. For example, the Toughness trait must be calculated so that we know what the bounding range can be. This means we'll need to do the bounding after "trtFinal" is calculated. We have a chicken-and-egg problem.

The bottom line of all this is that we really shouldn't consider using the "trtUser" field for our purposes with derived traits.

ALTERNATE FIELD FOR USER ADJUSTMENT

The good news is that we can simply create a different field for our needs and proceed with our original plan. Our new field will behave very similarly to "trtUser", so we'll start by cloning it. The field is only needed for derived traits, so we'll add it to the "Derived" component.

We can re-use the "trtMinimum" and "trtMaximum" fields for bounding, and we won't need to do any special handling of the bounds. We want to display the final value within the incrementer, which we can pull from the "trtFinal" field. We do not want to use the "trtDisplay" field, because the that field may contain secondary movement (e.g. "6/10" for burrowing) and it won't fit within the incrementer. Lastly, we want to allow the user to edit that value directly, so we'll include delta handling for our field. Our resulting field should look like the following.

```
 <field
   id="trtUserCre"
   name="User Value for Creature"
   type="user"
   defvalue="0"
   usedelta="yes"
   maxfinal="50">
   <!-- Bound the user value to the limits established for the trait
-->
   <bound phase="Traits" priority="5500" name="Bound
trtUserCre">
     <before name="Derived trtFinal"/><![CDATA[
     @minimum = field[trtMinimum].value
     @maximum = field[trtMaximum].value
     ]]></bound>
   <!-- Display the final calculated value to the user -->
   <finalize><![CDATA[
     @text = field[trtFinal].value
     ]]></finalize>
   </field>
```

With the field in place, we need to set it up properly. To accomplish this, we'll revise the Creation script for the "Creature" component. In addition to setting up the attributes, we must also assign the

initial user values for the derived traits. This consists of adding the following lines of code to the script.

```
~assign the appropriate adjustment values to derived traits
hero.child[trPace].field[trtUserCre].value = field[crePace].value
hero.child[trParry].field[trtUserCre].value = field[creParry].value
hero.child[trTough].field[trtUserCre].value =
field[creTough].value
hero.child[trCharisma].field[trtUserCre].value =
field[creCharis].value
```

The next step is to include the "trtUserCre" field value in the calculation of the final value. For this, we need to modify the Eval script that calculates "trtFinal" within the "Derived" component. After the value is calculated normally, we can change the code to use in the "trtUserCre" field for creatures instead of the "trtCreatur" field, as shown below.

```
~if this is a creature, we need to add the user value as a custom
adjustment
if (hero.tagis[Hero.Creature] <> 0) then
  field[trtFinal].value += field[trtUserCre].value
  endif
```

Hookup User Manipulation

At this point, we have the basic internal workings in place. Before we spend more time here, let's see how things work within the interface. We need to expose the field value for change by the user. This entails modifying the "baTrtPick" template that is used to show derived traits on the "Basics" tab. If we want the user to modify the value, we need to add a new incrementer portal. We can use a simple incrementer style that is provided by the Skeleton files, which results in the new portal shown below.

```
<portal
  id="value"
  style="incrSimple">
  <incrementer
    field="trtUserCre">
    </incrementer>
  <mouseinfo><![CDATA[
    @text = "Adjust this trait by clicking on the arrows to
increase/decrease the value assigned."
    ]]></mouseinfo>
  </portal>
```

Within the Position script, we need to do two things. First, we must make sure that we only show either the new incrementer or the old "details" portal, which means controlling visibility based on whether we have a creature or not. This can be done at any point in the script. Second, we need to center our new incrementer in the same general region used by the existing "details" portal so that it occupies the same region. We have to do this after the "details" portal is positioned. This results in the following code being added to the Position script.

```
~the incrementer is visible if we have a creature, else the details
if (hero.tagis[Hero.Creature] <> 0) then
  portal[details].visible = 0
else
  portal[value].visible = 0
  endif

~center the incrementer over the details portal
perform portal[value].centeron[horz,details]
perform portal[value].centervert
```

We're now ready to give our changes a try. When we create a new creature, our incrementers show up where they should and have appropriate initial values displayed. We should be able to make adjustments via the incrementers and see the corresponding values change for the creature.

Bounding the Derived Traits

At this point, we've got a user-modifiable value that will be setup properly and factored into the final adjustment calculation. The final piece we're missing is the bounding, since the user can freely modify the value to silly numbers. We'll achieve this by setting up appropriate values for "trtMinimum" and "trtMaximum", ensuring that the user can adjust the starting value within reason.

In order to properly bound the value, we need to know the original starting value, since we'll establish our limits relative to that value. We also need to know what the absolute minimum value is for a given trait (i.e. its "floor" value). This value differs for each derived trait, since Charisma can go negative, Pace can't drop below one, and the others can't drop below two.

We already have the starting value in the "trtCreatur" field. However, we need to introduce a new field on the "Derived" component to track the "floor" value. Since this value will be setup appropriately by the creature, it is a simple derived value. The new field should look like below.

```
<field
  id="trtFloor"
  name="Creature Floor"
  type="derived">
  </field>
```

We need to setup the "floor" value every evaluation cycle, just like we're already doing for the "trtCreatur" field. This is done by augmenting the existing Eval script on the "Creature" component. We simply need to assign the appropriate values to the field for each derived trait, which is accomplished by adding the following lines of code to the script.

```
~setup the proper floor values for each trait
hero.child[trPace].field[trtFloor].value = 0
hero.child[trParry].field[trtFloor].value = 2
hero.child[trTough].field[trtFloor].value = 2
hero.child[trCharisma].field[trtFloor].value = -4
```

We can now utilize these values for properly bounding the "trtUserCre" value. We'll define a new Eval script on the "Derived" component for this purpose. Our minimum and maximum will range from -3 to +3 from the original starting value. We'll also use the floor value to verify that we don't let the user drop a trait below its absolute minimum. The only special detail about this script is its timing. We have to schedule this script after all standard traits are calculated and before the Bound script is evaluated on the "trtUserCre" field. This results in the new Eval script below.

```
<eval index="3" phase="Traits" priority="5300">
  <before name="Bound trtUserCre"/><![CDATA[
  ~setup a minimum at 3 below the starting value; if our
minimum will yield a
  ~value below the minimum for this trait, limit it to the minimum
  field[trtMinimum].value = field[trtCreatur].value – 3
  var bonus as number
```

```
    bonus = field[trtBonus].value + field[trtInPlay].value
    if (field[trtMinimum].value + bonus < field[trtFloor].value) then
      field[trtMinimum].value = field[trtFloor].value – bonus
      endif

   ~now setup a maximum at 3 above the starting value
    field[trtMaximum].value = field[trtCreatur].value + 3
    ]]></eval>
```

Reload the data files and use our test creature. Let's attempt to adjust the derived traits. Decreasing Parry stops us at "2", since we set that up as our absolute floor. However, we can increase Parry a total of three notches. Our Toughness defaults to "4" and has the same behaviors. Our Pace is limited to a range of "2" to "8". Everything checks out.

Setup the Delta

All of our behaviors are in place, except for one. Derived trait values are simple numbers and not die types. Consequently, we want to let the user edit the value directly within the incrementer where they are shown. However, we want the user to edit a value that makes sense to him (i.e. the final result) instead of the adjustment value. To do this, we need to set the "delta" for the "trtUserCre" field to the difference between the actual user value and the value the user will see. Once that's done, HL will handle the rest for us. The new Eval script should look like the following.

```
  <eval index="4" phase="Render" priority="5000"><![CDATA[
     if (hero.tagis[Hero.Creature] <> 0) then
       field[trtUserCre].delta = field[trtBonus].value +
  field[trtInPlay].value +   herofield[acNetPenal].value
       endif
     ]]></eval>
```

We should also verify our delta is working properly. Reload the data files and select our test creature. Click within the incrementer showing the Parry and it should show the current value for editing. Enter the new value "4" and everything appears to work fine. Now try entering a value of "9". The new value is automatically bounded to the maximum we established of "6" (i.e. our initial value of 3 plus and additional 3). Derived traits are fully operational.

Let User Add Abilities

We allow the user to customize creature traits from the defaults. It would optimal if we allowed the user to do the same for creature abilities.

Dynamic Abilities Table

The first step in this process is to convert the table of abilities to be dynamic. We only want to show common abilities that are used across multiple creatures, so we'll leverage the "User.Creature" tag we defined earlier for this purpose. The changes are simple and result in the new portal below.

```
<portal
  id="abAbility"
  style="tblNormal">
  <table_dynamic
    component="RaceAbil"
    showtemplate="SimpleItem"
    choosetemplate="SimpleItem">
    <candidate>User.Creature</candidate>
    <titlebar><![CDATA[
      @text = "Add a Monstrous Ability"
      ]]></titlebar>
      @text = "Monstrous Abilities"
      ]]></headertitle>
    <additem><![CDATA[
      @text = "Add Monstrous Abilities"
      ]]></additem>
    </table_dynamic>
  </portal>
```

Customizable Abilities

This works great, but it doesn't handle abilities that need to be customized. For example, the "Immunity" ability needs to specify the nature of the immunity, while the "Size" ability needs to specify the rating adjustment. We need to let the user customize these abilities properly.

In order to do that, we need to identify the abilities that can be customized. We also need to identify how they can be customized. We can define a pair of tags for this purpose. The approach will be similar to the way that domains are handled, so we'll have one tag to identify when a value is needed and another to identify when text is needed. The two tags will be defined in the "User" tag group and should look like the following.

```
  <value id="NeedText"/>
  <value id="NeedValue"/>
```

With the tags defined, we can go back to all of our abilities and flag them appropriately. The assumption we'll make is that no ability requires both a value and text - always one or the other.

We now need to expose the customization fields to the user within the table. This requires that we define a new template for the purpose. Since our needs a similar, we can clone the template used for skills and adapt it. We're going to need the usual "name", "info", and "delete" portals. In addition, we're going to need one edit portal for entering text and another for values, plus a label to show next to the edit portal. We'll use a Label script to tailor the label based on whether a value or text is required. For the name, we need to show the customized name if the pick is not user-added or the original thing name if we're showing the portals to customize the ability. In the Position script, we only show one of the two edit portals, and we only show it if the pick has been added by the user (i.e. can be deleted). This results in the template presented below.

```
<template
  id="abPick"
  name="Ability Pick"
  compset="RaceAbil"
  marginhorz="3"
  marginvert="2">

  <portal
    id="name"
    style="lblNormal"
    showinvalid="yes">
    <label>
      <labeltext><![CDATA[
        if (isuser = 0) then
          @text = field[name].text
        else
          @text = field[thingname].text
          endif ]]></labeltext>
      </label>
    </portal>

  <portal
    id="label"
```

```
    style="lblSecond">
    <label>
      <labeltext><![CDATA[
        if (tagis[User.NeedText] <> 0) then
          @text = "Details:"
        else
          @text = "Rating:"
        endif ]]></labeltext>
      </label>
    </portal>

  <portal
    id="text"
    style="editNormal"
    width="100">
    <edit
      field="abilText">
      </edit>
    </portal>

  <portal
    id="value"
    style="editNormal"
    width="25">
    <edit
      field="abilValue"
      format="integer"
      signed="yes">
      </edit>
    </portal>

  <portal
    id="info"
    style="actInfo">
    <action
      action="info">
      </action>
    <mouseinfo/>
    </portal>

  <portal
    id="delete"
    style="actDelete"
    tiptext="Click to delete this item">
    <action
      action="delete">
      </action>
    </portal>

  <position><![CDATA[
    ~set up our height based on our tallest portal
    height = portal[info].height

    ~if this is a "sizing" calculation, we're done
    if (issizing <> 0) then
      done
      endif

    ~position our tallest portal at the top
    portal[info].top = 0

    ~determine whether our user fields are visible
    ~Note: If the pick cannot be deleted, it has been
bootstrapped, so we assume
    ~       that no users fields can be specified
    portal[value].visible = 0
    portal[text].visible = 0
    if (candelete <> 0) then
      if (tagis[User.NeedValue] <> 0) then
        portal[value].visible = 1
      elseif (tagis[User.NeedText] <> 0) then
        portal[text].visible = 1
        endif
      endif
    if (portal[value].visible + portal[text].visible = 0) then
      endif

    ~position the other portals vertically
    perform portal[name].centervert
    perform portal[delete].centervert
    perform portal[label].centervert
    perform portal[text].centervert
    perform portal[value].centervert

    ~position the delete portal on the far right
    perform portal[delete].alignedge[right,0]

    ~position the info portal to the left of the delete button
    perform portal[info].alignrel[rtol,delete,-8]

    ~position the name on the left
    portal[name].left = 0

    ~position the label and edit portals to the right of the name
    perform portal[label].alignrel[ltor,name,15]
    perform portal[value].alignrel[ltor,label,3]
    portal[text].left = portal[value].left

    ~if the ability is auto-added, change its font to indicate that
fact
    if (candelete = 0) then
      perform portal[name].setstyle[lblAuto]
      endif
    ]]></position>
  </template>
```

Surrogate User Fields

Unfortunately, our new template won't compile. The problem is that we can't associated "derived" fields with edit portals - only "user" fields are allowed. So we'll change the two fields to be "user" fields. However, this creates a new problem, as we can't assign values via bootstraps to "user" fields on unique picks. We could change all of our abilities to be non-unique, but then the user could assign the any ability multiple times, even when doing so is meaningless. That's not a good option.

We seem to be at a stalemate, so it's time to get creative. We need "user" fields to work with the edit portals, and we need "derived" fields to be assigned via bootstraps. It looks like the solution is to have both. We'll start by defining a pair of new fields on the "RaceAbil" component, as shown below.

```
  <field
    id="abilUsrVal"
    name="User Value"
    type="user">
    </field>

  <field
    id="abilUsrTxt"
    name="User Text"
    type="user"
    maxlength="25">
    </field>
```

Once we hook up the edit portals to these two fields, the compiler is happy. We can reload our files and actually enter values for abilities that we add. Now we need to connect our "user" fields with the "derived" fields that are used by the Eval scripts on the various abilities.

We can accomplish this by defining an Eval script on the "RaceAbil" component. This script will copy the contents of the "user" fields into the corresponding "derived" fields. Since "user"

fields are always in place from very beginning, we'll schedule the script to occur early in the evaluation cycle, ensuring that the "derived" fields contain the proper values when accessed. The one thing we need to be careful of, though, is that we must not copy the field values if the user did not add the pick. If we do, then we'll overwrite the values assigned via the bootstraps with empty values, since no user values will exist for those abilities. Putting this all together yields the Eval script shown below.

```
<eval index="3" phase="Setup" priority="5000"><![CDATA[
  ~if this pick was NOT user-added, no user fields can be
accessed
  if (isuser = 0) then
    done
    endif

  ~if we have a user value, put it into the derived field
  if (tagis[User.NeedValue] <> 0) then
    field[abilValue].value = field[abilUsrVal].value

  ~if we have user text, put it into the derived field
  elseif (tagis[User.NeedText] <> 0) then
    field[abilText].text = field[abilUsrTxt].text
    endif
  ]]></eval>
```

With the script in place, we can reload the data files and everything works the way we want it. We can easily mix and match bootstrapped abilities with user-added abilities, and the user can properly customize any abilities that he adds.

THE FINAL STAGES

FINAL CLEANUP (SAVAGE)

Our data files are complete. It's time to do a final round of cleanup. We'll be eliminating any extraneous mechanisms that we inherited from the Skeleton files. After that, we'll go through and implement every individual thing in the rulebook. This includes the complete list of edges, hindrances, skills, weapons, gear, etc. Lastly, we'll do one last round of testing so we can fix anything that has slipped past us.

SKELETON FILE REMNANTS

The Skeleton files provided us with a working set of functionality, and we used the vast majority of it. However, there are some bits left over that we didn't use. Before we consider our files complete, we should prune that material out, leaving ourselves with only material that is associated with our game system.

WARNING! Deleting information from the data files will result in lots of warnings about orphaned material when you load saved portfolios. This is normal and can be ignored, since we are intentionally deleting the information. If you encounter errors like this when you aren't intentionally deleting material, be sure to verify that you haven't made an inadvertent mistake.

We'll focus our attention first on the various things that are defined. That's because any internal mechanisms we might eliminate or change will impact these things, so it's much easier to remove the dependencies before removing the mechanisms that are depended upon. All of these things will be defined within the data files that start with the prefix "thing_". The Skeleton files included a variety of thing definitions that we're no longer using. Those we can readily identify and delete.

We'll start with any things defined for the purpose of validation. Using the logic above, validation rules often build on other things, so we eliminate them before eliminating the things they depend on. All validation rules are found in the file "thing_validate.dat", so we'll go through the each thing defined in the file and determine if we're still using it. There appears to be only one, the "valCP" thing used for testing character points. We can delete that and move on.

Proceeding down the dependency hierarchy, the next group of things to focus on are any resources or trackers, which are defined in the file "thing_miscellaneous.dat". There are two resources in this file that we inherited from the Skeleton files are no longer using. These are the "resCP" and "resAbility" things, so we'll delete them.

Once we do that, we'll discover a couple of old dependencies on those two things within the "resXP" thing. We're definitely using the "resXP" resource, but the Eval script that adjusts the available quantity of those resources based on the XP is extraneous. Once we delete that, this file is cleaned.

However, there is a second dependency. The "Ability" component we inherited from the Skeleton files has an Eval script that accrues the usage of ability points. With the elimination of the "resAbility" thing, we must also eliminate the script.

We can now go through all the other data files in which we defined things. Our goal is to eliminate any sample things that were provided or that we copied, adapted, and left hanging around. The list below spells out the various places we need to check.

- Sample attributes
- Sample skills
- Sample abilities
- Sample adjustments
- Sample advances
- Sample weapons
- Sample armor
- Sample gear
- Sample races
- Sample edges
- Sample hindrances

After going through the data files in search of the above items, there should only be one file left that we haven't assessed and in which things are defined. That's the file "thing_traits.dat". The Skeleton files provided us with a number of traits that we didn't use, and there should no longer be any references to them. Consequently, we can now delete them easily. The set of things to remove are "trHealth", "trInit", "trDefense", and "trPowerPts".

Our data files should no longer have extraneous things lurking within them, so we can shift our focus to the internals. In general, we don't really need to worry much about mechanisms built into the various components. Most of those mechanisms have been fully adapted to our purposes for Savage Worlds. The few exceptions are causing no trouble and will require a fair amount of work to isolate and remove. So it's best to just leave them alone, unless they are either obvious or a source of potential problems.

In our case, there are two places that are worth addressing. The first is the "Ability" component. We changed this component so that it provides a number of shared behaviors, but the Skeleton files originally defined it without that in mind. As such, there is an important behavior provided by the component that is either redundant or a potential problem. The component defines an identity tag group and assigns a corresponding tag to the hero for

every ability. Each separate type of ability already does this with its own identity group, so we can delete the identity group definition and the Eval script that forwards the tag to the actor.

The final cleanup task we should do is review the contents of the "Actor" component. For many game systems, at least a few mechanisms provided in this component will be unused. Since this component is rather extensive, keeping it pruned of unnecessary logic will simply make our data files easier to maintain. We can go through the list of fields and identify those that are now orphaned. This consists of the fields "acStartCP" and "acStartAbi", which we can delete. Scanning through the rest of the component, everything else is being used.

At this point, our data files have been properly cleaned of any unused remnants of the Skeleton files.

NOTE! If our game system did not use the advancements mechanism, we could also delete the data files specific to advancements and all the associated references.

IDENTIFYING TIMING PROBLEMS

One of the details we should verify during cleanup is whether we have any timing problems within our data files. Unless we did an exhaustive job of establishing and maintaining timing dependencies during development (we didn't), it's quite likely that we're going to have at least one or two holes in our logic. These holes might not even be visible right now, but future changes or user extensions could expose them.

The best thing we could do right now is to go through and establish thorough timing dependency relationships everywhere. Any task that relies on other values can be assigned a named "after" timing dependency on the scripts that calculate those values. Alternately, the scripts that calculate can be assigned a named "before" dependency on the script that uses the value. That's going to entail a healthy amount of work, so it's probably not a practical solution.

What we can do instead is some spot testing to see if we can uncover problems. Whenever we do uncover a problem, we can take the time to add the appropriate timing dependency relationships to the involved tasks. This won't be perfect, but it will be much faster and will ensure that we begin improving our use of enforced timing dependency relationships.

SAMPLE TIMING PROBLEM

After a bit of poking around, there is one problem we can uncover. If we change the default value of attributes from two to zero, we'll then rely on the proper bounding logic to be applied that keeps the value between two and six. However, making that change reveals a problem that we need to resolve.

When we create a new character, the number of attribute points is being displayed as "-12 of 5" above the table of attributes. The moment that we make any change at all to the character, everything behaves correctly. We obviously have a timing problem.

The value "-12" is quite suspicious, too. We have a total of six attributes defined (including the hidden one for super powers). That translates to a difference of two points per attribute. It appears that we broke this when we changed the default value of each attribute from two to zero.

The attribute values are being properly bounded to two, as evidenced by everything being corrected by making any change to the character. A change triggers a new evaluation pass. This means that the field on the resource that contains the message for display is probably being synthesized before the bounding of the field value is occurring. Unfortunately, the message is being synthesized during the Render phase, so that's not the case.

Looking a bit more closely, the message being synthesized is using the "resLeft" field. That field is being calculated at a timing of Final/1000. However, the bounding is occurring at a timing of Final/10000. We've uncovered the source of the problem.

We need to change the timing of one or both actions. Before we can do that, though, we must first assess what other timing dependencies exist on these two actions. The calculation of the "resLeft" field is named, so we need to assess what other scripts depend on it. There are two, but both of them must occur before the calculation, so we are free to move the calculation later if we wish.

The bounding script is also named, and there is one dependency. The calculation of the delta must occur before the bounding is performed, and the delta depends on the fields "trtBonus" and "trtInPlay". Both of those fields must be in their final state before they are used to calculate the "trtFinal" value, and that happens at a timing of Traits/3000. This means that we can freely move the delta calculation to substantially earlier if we wish.

The pieces of the puzzle are now in place. The best solution is probably to change the timing of all three scripts. We'll calculate the delta for "trtUser" at a timing of Final/1000. Then we can move the timing of the bounding to Final/5000. Lastly, we can move the timing of the "resLeft" calculation to Final/7000. We'll also specify an "after" timing dependency of the "resLeft" calculation upon the bounding so that we don't run into this problem again in the future.

Once we make these changes, everything should work properly again. Unfortunately, the problem remains. We found a piece of the problem, but there is something else amiss still.

After adding some "debug" statements to our scripts, we'll confirm that "resLeft" calculation is yielding the wrong results. So we need to focus on the information that the "resLeft" calculation depends upon. This would be the "resMax" value and the "resSpent" value. Doing a quick search of where the "resMax" field is referenced in the data files confirms that the "resMax" value is not the culprit, so that leaves the "resSpent" field. Doing another scan of where the "resSpent" field is referenced uncovers that it is being tallied for attributes via an Eval script at a timing of Setup/5000. That's way before the bounding occurs, which explains the problem. If we change the timing to Final/6000 (i.e. after the bounding), everything starts to work correctly.

To make sure this problem doesn't arise again, we'll add a timing dependency to the erroneous Eval script. We'll ensure that this script always occurs after the bounding of the "trtUser" field.

But wait a minute. The script we fixed only applies to attributes. A similar script exists to tally the points spent for skills. Since skills are traits, they also need to be tallied after the bounding is performed. Skills start with a default "trtUser" value of zero, so we aren't noticing the error. However, if we changed the default value, we would see the same problem. The solution is to make sure that the Eval script on the "Skill" component is also changed to the same timing and that a suitable timing dependency is also added.

The only other use of the "Trait" component is with derived traits. Since there is no resource being tallied for derived traits, there are no other places where this problem could be lurking.

Once the bug is resolved, we can restore the original default value back to two.

Rank and XP Shown for Creatures

There are a number of places where XP and rank are being shown for creatures. We eliminated this from a few obvious spots, but we overlooked some as well. We need to go through these locations and verify that the XP and rank are omitted for creatures. We can identify all the places that we need to check by doing a search for references to the "acRankName" field name.

The first instance we find is within the LeadSummary script in the definition file. We can wrap the logic for including the XP within an if/then block, as shown below.

```
~append the rank and XP (only for non-creatures)
 if (hero.tagis[Hero.Creature] = 0) then
   @text &= " - " & herofield[acRankName].text & " - " &
herofield[acFinalXP].value & " XP"
   endif
```

The next instance is within the Eval script that synthesizes the ally recap within the "Actor" component. We can use the same technique and wrap the code within an if/then block. However, this instance requires a little more tweaking than just an if/then block, since the punctuation output assumes the XP is always included. The revised logic should look like the code below.

```
~output any race
 recap &= field[acRaceName].text

~output the XP and rank (only for non-creatures)
 if (hero.tagis[Hero.Creature] = 0) then
   recap &= ", " & field[acRankName].text & "  (" &
field[acFinalXP].value & " XP)"
   endif
```

There are two instances within character sheet output that need to be addressed. One is on the first page, where the character details are synthesized in the "oHeroInfo" portal. We can again wrap the logic within an if/then block, as shown below.

```
~append the rank and XP (only for non-creatures)
 if (hero.tagis[Hero.Creature] = 0) then
   @text &= "; " & herofield[acRankName].text & "  ("
   @text &= herofield[acFinalXP].text & " XP)"
   endif
```

The other instance within the character sheet output is on the second page, within the output for allies. We can use an if/then block again, although we need to be sure to keep the newline outside of the conditional block. The resulting code is below.

```
~output the rank and XP (only for non-creatures)
 if (hero.tagis[Hero.Creature] = 0) then
   @text &= "; {b}" & herofield[acRankName].text & "{/b} ("
   @text &= herofield[acFinalXP].text & " XP)"
   endif

~insert a newline before continuing our output
 @text &= "{br}"
```

The XP and rank are also shown within the "Basics" and "Journal" tabs. However, we have already dealt with both. On the "Basics" tab, the portal with the information is hidden for creatures. Similarly, the entire "Journal" tab is hidden for creatures.

Derived Traits Positioning Within Sheets

It turns out we didn't properly anticipate some of the nuances of derived traits when we first implemented character sheet output. All of the portals are positioned on a left-to-right basis within the "oDerivPick" template. However, if a derived trait is more than one character in width, everything ends up being aligned oddly.

We need to change the behavior so that the "adjust" portal is placed against the right edge. Once that's done, we can then place the "value" portal relative to the "adjust" portal on a right-to-left basis. This will ensure that everything always lines up consistently, regardless of the width of the "value" portal.

We can accomplish this change by replacing the code within the Position script that handles horizontal placement. While we're at it, we'll also assign a "marginhorz" attribute of "25" to the template, which allows us to simply place both the "name" and "adjust" portals flush against their respective edges. The revised code for the Position script is shown below.

```
~position everything horizontally
 portal[name].left = 0
 perform portal[adjust].alignedge[right,0]
 perform portal[value].alignrel[rtol,adjust,-20]
```

Show XP and Rank on Basics Summary Panel

There is plenty of available space at the bottom of the "Basics" summary panel, and we should consider putting that space to good use. One thing that would be potentially helpful on the panel is the character's current rank and XP. Adding it is easy, so let's do that now.

To keep things as simple as possible, we define a single label portal for displaying the rank and XP. We'll use the same approach as we did on the "Basics" tab. In fact, we might as well clone the "baRank" portal from that tab and adapt for the summary panel. After we change the unique id, the only detail we need to modify is the style. All other particulars remain the same, resulting in the new portal shown below.

```
<portal
  id="smRank"
  style="lblSummary">
  <label>
    <labeltext><![CDATA[
      @text = herofield[acRankName].text & "  (" &
herofield[acFinalXP].text & " XP)"
      ]]></labeltext>
    </label>
  </portal>
```

We now need to integrate the portal into the layout. We can add a "portalref" element and then simply auto-place the new portal beneath the "status" portal. In keeping with the policy we instituted a little bit ago, we also need to make sure that the new portal is not visible when the character is a creature. The revised layout should look like the following.

```
<layout
  id="smBasics">
  <portalref portal="smAttrib"/>
  <portalref portal="smDerived"/>
  <portalref portal="smStatus"/>
  <portalref portal="smRank"/>

 <position><![CDATA[
   ~the rank is only visible for non-creatures
   if (hero.tagis[Hero.Creature] <> 0) then
     portal[smRank].visible = 0
     endif

   ~position and size the tables to span the full layout
   perform portal[smAttrib].autoplace
   perform portal[smDerived].autoplace[20]
   perform portal[smStatus].autoplace[20]
   perform portal[smRank].autoplace[20]
   ]]></position>

 </layout>
```

TAILOR DISPLAY WIDTHS

There are a number of dynamic tables that present choices with lengthy descriptions. In order to better accommodate these descriptions, we need to increase the space afforded for showing those descriptions. This is controlled through the "descwidth" attribute on the table portals. The following changes should be made.

Skills table - increase to 350
Edges table - increase to 350
Hindrances table - increase to 350
Abilities table - increase to 350
Race chooser - increase to 350
Creature chooser - increase to 350

There are also a small number of other places where we can tweak the widths to better accommodate the information being displayed. These include the following:

Widen the "stRace" chooser on the static form by 10 pixels to accommodate all races

Widen the "arWpnThing" template on the "Armory" tab by 25 pixels

Widen the "arDefThing" template on the "Armory" tab by 25 pixels and shift the defensive value over the same 25 pixels

Widen the "adjust" portal within the "abAttrPick" template on the "Basics" tab by 10 pixels so that a bonus of "+10" (for creatures) doesn't get clipped. This requires also changing the alignment gaps of both the "adjust" and "value" portals to "-8".

Widen the "adjust" portal within the "skPick" template on the "Skills" tab by 5 pixels so that a bonus of "+10" doesn't get clipped. This requires also changing the alignment gap of the "adjust" portal to "6".

Widen the "attack" portal within the "smWeapon" template on the "Armory" summary panel by 10 pixels to show attacks better than "d12" (e.g. "d12+1")

CONSISTENT TERMINOLOGY WITH RULEBOOK

The Savage Worlds rulebook sometimes uses terminology that is different from what the Skeleton data files provide. When that occurs, we need to be sure to use the terminology that is consistent with the rulebook. We neglected to do that on the "Skills" tab, where we use the term "domain" instead of "focus".

We can easily change the label within the "skPick" template to show the term "Focus". However, if we modify the "Domain" component directly, we'll also change the behavior for domains with other game elements, such as hindrances. The proper way to customize the behavior of the "Domain" component is by defining a new tag within the "DomainTerm" tag group that has the proper term to be used. The new tag is shown below and must be defined within the "DomainTerm" tag group.

```
<value id="Focus"/>
```

We can then assign the tag to the "Skill" component, which results in all skills inheriting the proper term, as shown below.

```
<tag group="DomainTerm" tag="Focus"/>
```

There is one other place where we need to handle this properly. The advancements mechanism also prompts for domains, but it doesn't use skills directly. The various advancement things are used instead. So we need to assign the same tag to the advancement thing used for adding new skills (i.e. "advSkill").

After making these changes, we can reload the files and verify that all references to the term "domain" are correctly being mapped to "focus" for skills.

DESCRIPTION TEXT FOR HINDRANCES

The description text shown for hindrances should include the appropriate minor or major designation. The best way to address this is to handle it within the "Descript" procedure. By defining a new procedure for handling the custom aspects of hindrances, the "Descript" procedure can call the new procedure and synthesize everything properly

The new procedure must distinguish between things and picks. For picks, any user-selected severity must be shown. For things, the fact that a hindrance has a user-selectable severity must be indicated. The new procedure for hindrances is shown below. This procedure should be called when the "component.Hindrance" tag is encountered.

```
 <procedure id="InfoHinder" context="info"><![CDATA[
   ~declare variables that are used to communicate with our
caller
   var iteminfo as string
   iteminfo = ""

  ~center the severity rating
   iteminfo &= "{align center}"

  ~if the severity is user-selected, report that fact; otherwise,
report
   ~whether the hindrance is major or minor
   if (!ispick + tagis[User.UserSelect] >= 2) then
     iteminfo &= "(Minor or Major)"
   elseif (field[hinMajor].value <> 0) then
     iteminfo &= "(Major)"
   else
     iteminfo &= "(Minor)"
     endif

  ~terminate the line and stop the centered alignment
   iteminfo &= "{br}{align left}"
```

```
]]></procedure>
```

Description Text for Arcane Backgrounds

Arcane backgrounds a conferred via the corresponding edge. Consequently, the arcane background itself is never shown anywhere, so we don't provide the user with any of the details regarding the various arcane backgrounds. We should display that the details of each arcane background within the description for the pertinent edges.

One solution would be to enter all the details for each background into the description for the edge. However, it would be much better to maintain the description text for the arcane backgrounds separately and pull that text into the description for the edge. We can accomplish this with a little bit of scripting.

We'll define a new procedure that synthesizes the description particulars of an edge. Then we can call this procedure from within the "Descript" procedure when an edge is being rendered. Within our new procedure, we can use a "foreach" statement to access the bootstraps assigned to the edge. We'll only access the arcane background things, and we'll integrate their description text appropriately. We can also use a nested "foreach" statement that pulls the details of any drawback associated with the arcane background. The resulting procedure should look like the one below.

```
 <procedure id="InfoEdge" context="info"><![CDATA[
   ~declare variables that are used to communicate with our
caller
   var iteminfo as string
   iteminfo = ""

  ~if this edge does not possess an arcane background, there's
nothing special to do
   if (tagis[Arcane.?] = 0) then
     done
     endif

  ~output the particulars of the arcane background
   foreach bootstrap in this where "component.Arcane"

    ~append the description of the arcane background
     iteminfo &= eachthing.field[descript].text & "{br}"

    ~append any drawback associated with the arcane
background
     foreach bootstrap in eachthing where
"component.Drawback"
       iteminfo &= "{br}{b}" & eachthing.field[name].text & ":{/b} "
       iteminfo &= eachthing.field[descript].text & "{br}"
       nexteach

    nexteach
   ]]></procedure>
```

While we're in here, we should also identify the type of edge (e.g. background, social, etc.) within the description text. We can do that easily by inserting the following code before the check for an arcane background.

```
 ~report the type of edge
  iteminfo &= "Type: " & tagnames[EdgeType.?] & " Edge{br}"
```

Description Text for Races

In the same way that we included the arcane background and drawback details above, the display of races should present the list of racial abilities. We'll define a new procedure for this purpose and integrate it into the "Descript" procedure when a race is being handled. This time, we'll do some special formatting of headers above the racial abilities and the racial description text itself. The resulting procedure is shown below.

```
 <procedure id="InfoRace" context="info"><![CDATA[
   ~declare variables that are used to communicate with our
caller
   var iteminfo as string
   iteminfo = ""

  ~setup formatting details for re-use below
   var setup as string
   var wrapup as string
   setup = "{align center}{b}{i}{text 79cfcd}"
   wrapup = "{text 010101}{/i}{/b}{br}{align left}"

  ~output the particulars of each linked racial ability
   iteminfo &= setup & "- Racial Abilities -" & wrapup & "{vert 4}"
   foreach bootstrap in this
     iteminfo &= "{b}" & eachthing.field[name].text & ":{/b} "
     iteminfo &= eachthing.field[descript].text & "{br}{br}"
     nexteach

  ~introduce the race description
   iteminfo &= setup & "- Racial Description -" & wrapup & "{vert
-12}"
   ]]></procedure>
```

Identify Hindrance Severity in Sheet Output

The list of hindrances that are output within the character sheet use the same mechanism as edges and racial abilities. However, hindrances can be taken in both major and minor forms, and that aspect of each hindrance is important to the player.

We can modify the Label script currently in use within the "oAbilPick" template to detect the presence of a hindrance and output it specially. We'll only flag major hindrances, and we'll do it by including a special symbol. This will keep the space requirements to an absolute minimum. We can pick a suitable character from the "Wingdings" font, resulting in the following revised Label script.

```
 var major as string
  if (tagis[component.Hindrance] <> 0) then
    if (field[hinMajor].value <> 0) then
      major = " {font Wingdings}" & chr(181) & "{revert}"
      endif
    endif
  @text = field[shortname].text & major & "{/b}  {size 32}" &
field[summary].text
```

Add Drawbacks to Character Sheet

It seems that we overlooked including the details of arcane drawbacks on the character sheet. Since it's something the player should not forget, we should probably include a reminder. One solution would be to show the drawback like a hindrance. Another would be to include the name of the drawback within the title above the table of arcane powers. And a third would be to add a new table that listed the drawback just like an edge or hindrance.

None of these options is ideal, and they each have their trade-offs. Treating drawbacks as hindrances would entail a fair amount of

work to revise the data files. Since all we need is to display them, the extra work probably isn't worth it. Including the drawback within the title above the arcane powers table would not show its summary, plus it would result in a very cramped title area. Adding a new table is quick and easy, but the downside is that we'll consume additional vertical space on the sheet.

We'll go with the new table for the sake of simplicity. Our table portal can be adapted from the edges table, giving us the following.

```
<portal
  id="oDrawback"
  style="outNormal">
  <output_table
    component="Drawback"
    showtemplate="oDrawPick">
    <headertitle><![CDATA[
      @text = "Arcane Drawbacks"
      ]]></headertitle>
    </output_table>
  </portal>
```

The template can be readily adapted from the one used for special abilities. This results in the template shown below.

```
<template
  id="oDrawPick"
  name="Output Drawbacks Table"
  compset="Drawback"
  marginvert="2">

  <portal
    id="details"
    style="outMedLt">
    <output_label>
      <labeltext><![CDATA[ @text = field[name].text & "{/b}  {size
32}" & field[summary].text ]]></labeltext>
    </output_label>
  </portal>

  <position><![CDATA[
    ~our details width spans the entire template width
    portal[details].width = width

    ~our height is the height of our portal
    height = portal[details].bottom
    ]]></position>
  </template>
```

The last step is to integrate the portal into a layout. We'll add it to the "oRightSide" layout and place immediately above the table of arcane powers. If we include it above the powers, it will always be prominently visible and immediately adjacent to the list of powers. All we need to do is add the "portalref" element and then auto-place the portal immediately beneath the continuation of the edges output.

Ignoring Wound Penalties

There are two edges that allow the character to ignore wound penalties (Nerves of Steel and the improved version). Since HL automatically applies the effects of all wounds penalties to rolls, users will assume the effects of these edges are properly factored in. We'd better make good on that assumption.

The wound and fatigue penalties are all handled via the "acNetPenal" field on the "Actor" component. We can add a new field to track the number of wound penalties that are ignored, and then we can factor that value into the final calculation for "acNetPenal". Our new field is shown below.

```
<field
  id="acIgnWound"
  name="Ignored Wound Penalties"
  type="derived">
  </field>
```

We can now modify the Calculate script on the "acNetPenal" field to take the ignored wounds into account. The revised script should look like the following.

```
@value = -field[acFatigue].value
 if (field[acWounds].value > field[acIgnWound].value) then
  @value -= field[acWounds].value - field[acIgnWound].value
  endif
```

All the mechanics are in place. Our two edges can now use a simple Eval script to adjust the "acIgnWound" field, and everything works exactly as we want.

```
<eval index="1" phase="Setup" priority="5000"><![CDATA[
  herofield[acIgnWound].value += 1
  ]]></eval>
```

Dependencies on Load Limit

The "Acrobat" edge exposes a timing issue with our data files. The edge confers a +1 to a character's Parry if the character has no encumbrance penalty. To implement this, we need to have the encumbrance penalty determined before we act, and we need to act before the final value for the Parry trait is resolved. Our current script timing does not work this way, although we should be able to do this.

Our goal is to be able to adjust the Parry trait prior to its final resolution. Derived traits are officially calculated at a timing of Traits/6000. So we need to get the encumbrance penalty resolved before then if at all possible.

The encumbrance penalty is tracked by the "resEncumb" resource. The final value is currently calculated at a timing of Final/500. The only timing requirements for this calculation are the accrual of weight carried (which occurs at Effects/10000) and the load limit (which occurs at Traits/7000). The weight accrual occurs long before we need to act for the "Acrobat" edge, so we can move the timing of the penalty if we can move the timing of the load limit calculation.

The load limit is tracked by the "resLoadLim" resource and it calculates its maximum (the value we need for the encumbrance penalty) at a timing of Traits/7000. This calculation depends on the "acLoadMult" field, which is nailed down very early in the evaluation cycle, so that's not a problem. It also depends on the final value of the Strength attribute. Attributes are finalized within the "trtFinal" field, which is calculated at a timing of Traits/3000. This means we can move the load limit calculation to a timing of Traits/4000 without any problems.

Once we move the load limit calculation, we can then safely move the encumbrance penalty calculation. We'll move that to a timing of Traits/4500. This leaves nice gap in the evaluation cycle where we can inspect the encumbrance and adjust the Parry trait based on it. We'll schedule the Eval script for the edge at Traits/5000.

Since we've gone through and analyzed all these dependencies, we'll do one additional thing. Each of the various scripts we identified should be named. Then we can specify appropriate timing dependencies between these scripts. Now we can let the compiler safety check our timing dependencies for us all the time. If we need to adjust the timing of one of these scripts in the future, we can rely on the compiler to let us know if we messed something up.

CUSTOM DOMAINS FOR HINDRANCES

There are quite a few hindrances that utilize domains, and the domain for each is quite a bit different from the others. We currently just use the label "Domain" for each hindrance, but it would be great to customize the term for each one. We already used the "DomainTerm" tag on the "Skills" component, but we can also use the tag on individual things.

To support a varying domain term for each hindrance, all we need to do is modify the template that displays the hindrances and add the appropriate tag to each hindrance. Modifying the template simply requires us to change the "lbldomain" portal to use a Label script instead of a fixed string. If a "DomainTerm" tag is found, we use the term, else we default to the original "Domain:" label. The revised portal is shown below.

```
<portal
  id="lbldomain"
  style="lblSecond">
  <label>
    <labeltext><![CDATA[
      if (tagis[DomainTerm.?] <> 0) then
        @text = tagnames[DomainTerm.?] & ":"
      else
        @text = "Domain:"
        endif
    ]]></labeltext>
  </label>
</portal>
```

Since the "DomainTerm" tag group is dynamic, we don't need to define all the tags within the group. We can instead define them whenever we need them. So we can go through all the various hindrances and add a custom tag to each, which will then be used throughout HL. For example, the "Phobia" hindrance could be assigned a "Fear" domain term via the tag below.

```
<tag group="DomainTerm" tag="Fear"/>
```

EDGES NEED DOMAIN SUPPORT

When we implemented edges, we identified all the ones that looked to be special in some way and added appropriate handling for them. We missed one. The "Connections" edge requires that the user specify the organization with which the connections exist. This means we have to add domain support for edges. Fortunately, we've already done that a couple times, so it should not be complicated. In fact, it's rather simple.

The first thing we need to do is add the "Domain" component to the "Edge" component set. This will integrate domain handling internally into edges. The "Domain" component handles modifying the name, so all the mechanics we need are provided.

Next, we need to add support for domains to the interface. We can copy the two portals dealing with domains from the "edHinder" template and add them to the "edEdge" template. Then we can copy the positioning code for those portals, which we can drop into place with a single change. The "lbldomain" portal can be aligned directly relative to the "name" portal instead of using the non- existent "edge" variable.

Domain support is fully operational for edges now. All that remains to be done is assign the "User.NeedDomain" tag to the "Connections" edge. Once we do that, the edge will prompt the user for a domain and integrate into the name for display.

BACKGROUND EDGES ARE CREATION-ONLY

All background edges are generally only valid for selection during character creation. The Skeleton files provide built-in support for designating abilities as creation-only (via the "User.CreateOnly" tag). However, adding the tag individually to all background edges is error-prone. It would be best if we automatically treated all background edges as creation-only.

Doing this is easy. We can clone the pre-requisite used for this purpose within the "Ability" component and add it to the "Edge" component. We can then adapt it to only apply to background edges. Since GMs can allow the selection of background edges after creation, we'll only flag an error when showing things. Once an edge is added, we'll assume the player did so with the GM's approval. The resulting pre-requisite should look like below.

```
 <prereq message="Background edges are only available at
character creation">
    <!-- This pre-req is only applicable to background edges -->
    <match><![CDATA[
      EdgeType.Background
      ]]></match>

    <validate><![CDATA[
      ~we only report a failure on things (once added, we assume
the user knows best)
      if (@ispick <> 0) then
        @valid = 1
        done
       endif

      ~if the mode is creation, we're valid
      if (state.iscreate <> 0) then
        @valid = 1
        endif
      ]]></validate>
    </prereq>
```

BUYING OFF HINDRANCES

All the various advancements appear to be working, but, there's one situation we haven't dealt with yet. Although not in the official list of advancement options, the rules indicate that it is possible to spend an advance to "buy off" a hindrance. If a GM allows this option, we need to provide a way to handle it within the data files.

Unless the user adds a new advance, our validation logic will continue to report an error. So the easiest way to solve this is to add a new advance for the purpose of buying off a hindrance. This advance won't have any special behaviors, as it's really just a placeholder to consume a slot. However, we should allow the user to annotate the hindrance that was actually bought off and show it within the name of the advance.

Fortunately, the Skeleton files provide something simple like this that we can use. It's referred to as a "Notation" advance, which simply displays an edit field for the user to enter some text (such as

the name of the hindrance that was bought off). We can define a new advancement that leverages this mechanism, as shown below.

```
<thing
  id="advBuyOff"
  name="Buy Off Hindrance"
  compset="Advance"
  description="Description goes here">
  <fieldval field="advAction" value="Buy Off Hindrance"/>
  <fieldval field="advCost" value="1"/>
  <tag group="Advance" tag="Notation"/>
  <child entity="Advance">
   </child>
  </thing>
```

This works great, but a bit of testing reveals another problem. If the user takes a hindrance at character creation and then buys it off via an advance, he'll need to delete it. The entails switching back to creation mode, deleting, the hindrance, and then switching back. Unfortunately, once the hindrance is deleted, the number of rewards taken won't match the hindrances and the users won't be able to return to advancement mode.

The solution here is to define a new hindrance that serves as a placeholder for another hindrance that is bought off via an advance. This new hindrance can be added after the original hindrance is deleted and all the validation checks will be satisfied. This new hindrance must be fully customizable. We also don't want it to appear on the "Special" tab or within character sheet output. This results in the new hindrance shown below.

```
<thing
  id="hinBuyOff"
  name="– Buy Off –"
  compset="Hindrance"
  summary="Buy off a previous hindrance"
  description="Description goes here">
  <fieldval field="hinMajor" value="0"/>
  <tag group="User" tag="UserSelect"/>
  <tag group="User" tag="NeedDomain"/>
  <tag group="DomainTerm" tag="Hindrance"/>
  <tag group="Hide" tag="Special"/>
  <tag group="Print" tag="NoPrint"/>
  </thing>
```

INJURIES FOR NPCS

When we first added support for injuries, treating them as advancements made perfect sense. Unfortunately, with the support of NPCs, treating them as advancements doesn't work very well unless the character happens to be a PC. The reason for this is that advancements are not applicable to NPCs, so we hide the entire tab.

We need to allow injuries to be assigned to NPCs via some mechanism. Since advancements are intended for the application of permanent injuries, it would also be helpful to provide a convenient way to apply temporary injuries to PCs.

The first idea that comes to mind is to add the injuries as in-play adjustments. The problem with this approach is that we can only modify existing picks via adjustments. Menus do not let us add new picks to the character, so we would have to improvise something really creative in order to get injuries like "Hideous Scar" to work (i.e. to bootstrap the hindrance onto the character).

Adding a separate table of injuries to the "In-Play" tab would consume a good amount of space, and the tab is already pretty cramped. There might be plenty of room on a big screen monitor, but we have to be mindful that some users will have HL running on a small laptop at the game table. Consequently, we have to make sure everything will work on a relatively small screen.

Looking through everything, there is one convenient place where we could easily add injuries. On the "Personal" tab, there is a gap beneath the image gallery that would fit a small table of injuries very easily. We'll anchor the injury list to the bottom, immediately above the permanent adjustments list. That way, we can show as many character images as we have space for. For PCs, we'll clearly label the table as "Temporary Injuries", just to be safe.

There are a number of steps involved, but the first thing we'll do is create a new sort set. Our table of injuries will potentially exist in a very small space. When injuries are added to the table, they will be sorted alphabetically by default. What we want is to show the injuries with the most recent ones at the top. That way, the user will be able to easily identify wounds from the most recent combat. The Kit provides a special tag group with the unique id "_CreateSeq" that allows the sorting of items in the order they were created. By reversing this order, we can have a sort set that gives us exactly what we want, as shown below.

```
<sortset
  id="MostRecent"
  name="Reverse Chronological Order">
  <sortkey isfield="no" id="_CreateSeq" isascend="no"/>
  </sortset>
```

With the sort set in place, we'll define a table portal in which we can manage injuries on the "Personal" tab. We can easily use the "SimpleItem" template for showing the selected injuries, and we can use the "LargeItem" template when the user chooses injuries. The resulting table portal is quite simple should look like the following.

```
<portal
  id="peInjury"
  style="tblNormal">
  <table_dynamic
   component="Injury"
   showtemplate="SimpleItem"
   choosetemplate="LargeItem"
   showsortset="MostRecent">
   <titlebar><![CDATA[
    @text = "Select a New Injury from the List Below"
    ]]></titlebar>
   <headertitle><![CDATA[
    if (hero.tagis[Hero.PC] <> 0) then
      @text = "Temporary Injuries"
   else
      @text = "Injuries"
      endif
    ]]></headertitle>
   <additem><![CDATA[
    @text = "Add New Injury"
    ]]></additem>
  </table_dynamic>
 </portal>
```

Our final task is to integrate the new table portal into the layout on the "Personal" tab. We're inserting the portal between the images and adjustments tables, so we need to ensure that the tab order reflects that when we add our "portalref" element. After that, we can readily splice the logic for the new portal into the Position script for the layout. The revised logic that includes the impacted script code is presented below.

```
~reserve a width of 185 pixels for the user images table
 portal[peImages].width = 185

~use the same horizontal space for the injuries table
 portal[peInjury].width = portal[peImages].width

~position the injuries table immediately above the permanent
adjustments and
 ~reserve space for at least two items
 portal[peInjury].left = width - portal[peInjury].width
 portal[peInjury].maxrows = 2
 portal[peInjury].top = portal[peAdjust].top -
portal[peInjury].height - 10

~position the images table in the upper right corner
 portal[peImages].left = width - portal[peImages].width
 portal[peImages].height = portal[peInjury].top -
portal[peImages].top - 10

~if there is vertical space for more injuries, take advantage of it
 portal[peInjury].height = portal[peAdjust].top -
portal[peImages].bottom – 20
 portal[peInjury].top = portal[peAdjust].top -
portal[peInjury].height - 10
```

HIDING EQUIPMENT

Various creatures define their own custom equipment, such as the "Goblin" and "Lich". This gear needs to be private to the particular creature and not make public for general selection by the user. To solve this, we need to define a new "Hide.Equipment" tag and then assign that tag to all such gear.

Once the tags are assigned, we need to modify all of the table portals on the "Armory" tab. This new tag must be integrated into the Candidate tag expression of each portal so that equipment with the tag is not shown for selection. In each case, the revised element should look like the one shown below.

```
<candidate inheritlist="yes"><![CDATA[
   !Equipment.Natural &  !Hide.Equipment
  ]]></candidate>
```

SHOW LINKED ATTRIBUTES ON SKILLS

On the "Skills" tab, we neglected to show the linked attribute with each skill. While not critical, having the linked skill visible can be helpful. We should also include the linked attribute within the description text for each skill.

We'll start by modifying the description text. We can define a new procedure to synthesize the information for skills and integrate it into the "Descript" procedure when a skill is processed. All we need to do is add the one piece of data, which is solved with the simple procedure below.

```
<procedure id="InfoSkill" context="info"><![CDATA[
  ~declare variables that are used to communicate with our
caller
  var iteminfo as string
  iteminfo = ""

 ~report the linked attribute
  iteminfo &= "Linked Attribute: " &
linkage[attribute].field[name].text & "{br}"
  ]]></procedure>
```

There are two different places where we should show the linked attribute within tables. The first is within the list of selected skills that appears on the "Skills" tab. The other is within the table presented for the user to select skills to be added.

The linked attribute is a secondary piece of information, so it should be less prominent than the other facets of the skill. As such, we'll add it in a separate portal that appears at the far right and in a softer color. Within the "skPick" template that shows the skills added to the character, we can add the portal shown below.

```
<portal
  id="attribute"
  style="lblSecond">
  <label>
   <labeltext><![CDATA[
    @text = "(" & linkage[attribute].field[trtAbbrev].text & ")"
    ]]></labeltext>
  </label>
 </portal>
```

Once the portal is added, we can integrate it into the Position script for the template. There are a few things we need to tweak to complete the integration. The pertinent lines of the script are shown below.

```
~position the other portals vertically
 ...
 perform portal[attribute].centervert

...

~position the attribute portal to the left of the info button
 perform portal[attribute].alignrel[rtol,info,-8]

...

~position the name next to the adjustment
 perform portal[name].alignrel[ltor,adjust,8]

~if we don't need a domain, hide it and let the name use all
available space
 if (tagis[User.NeedDomain] = 0) then
  portal[lbldomain].visible = 0
  portal[domain].visible = 0
  portal[name].width =
minimum(portal[name].width,portal[attribute].left -
portal[name].left - 10)

~otherwise, position the domain portals next to the name
 else
  perform portal[lbldomain].alignrel[ltor,name,15]
  perform portal[domain].alignrel[ltor,lbldomain,2]
  portal[domain].width = minimum(150,portal[attribute].left -
portal[domain].left - 5)
  endif
```

Adding the linked attribute to the table for choosing skills requires a little bit more work. The "skSkills" table portal currently just uses the "SimpleItem" template for selection. In order to add the linked attribute, we'll need to create our own new template. We can do this easily by copying the "SimpleItem" template and calling it "skThing". Once that's done, we can modify the "skSkills" portal to reference our new template for choosing a skill.

The final thing we need to do is revise the template for our purposes. We can get rid of all the portals except for the name, then we can add a new portal similar to the one we added above for

showing the linked attribute. When we're finished, the new template should look similar to the one below.

```
<template
  id="skThing"
  name="Skill Thing"
  compset="Skill"
  marginhorz="3"
  marginvert="2">

  <portal
    id="name"
    style="lblNormal"
    showinvalid="yes">
    <label
      field="name">
      </label>
    </portal>

  <portal
    id="attribute"
   style="lblSecond">
    <label>
      <labeltext><![CDATA[ @text =
linkage[attribute].field[name].text ]]></labeltext>
      </label>
    </portal>

  <position><![CDATA[
    ~set up our height based on our tallest portal
    height = portal[name].height

    ~if this is a "sizing" calculation, we're done
    if (issizing <> 0) then
      done
      endif

    ~center the portals vertically
    perform portal[name].centervert
    perform portal[attribute].centervert

    ~position the attribute portal on the far right
    perform portal[attribute].alignedge[right,0]

    ~position the name on the left and let it use all available
space
    portal[name].left = 0
    portal[name].width =
minimum(portal[name].width,portal[attribute].left - 10)
    ]]></position>

  </template>
```

Hindrance Validation Incorrect

The validation logic we added for hindrances works great when there are only user-added hindrances on the character. The moment that new hindrances are added via advancements and/or injuries, the logic fails. The problem is that we forgot to take into account these other types of hindrances within the validation rule. Fortunately, we can easily detect they other types of hindrances. The code below shows the revised logic needed within the script for tallying up the hindrances we care about.

```
 ~iterate through all hindrances and tally up the number of
majors and minors
  ~Note: We must only tally hindrances that are user added and
not an advance.
  foreach pick in hero where "component.Hindrance"
    if (eachpick.isuser +  !eachpick.tagis[Advance.?] >= 2) then
      if (eachpick.field[hinMajor].value = 0) then
        major += 1
      else
        major += 1
      endif
    endif
  nexteach
```

Skill Points Must Ignore Advancement

Our skill points are tallying up fine during character creation, but we run into a problem once the character starts taking advances. The issue is that any attribute advances apply that advance to the "trtBonus" field. That's the same field we rely on to indicate when racial adjustments are applied to the character. The net result is that an attribute advance is being interpreted the same as a bonus at character creation, so our skill points tally becomes wrong once the advance is taken.

You might be wondering what the problem is with this, since the tracking of skill points isn't important once a character is created and enters advancement mode. The issue arises if the user decides to switch back to creation mode for some reason, make an adjustment, and then switch back to advancement mode. Once the user switches to creation mode, he won't be allowed to return to advancement mode due to the incorrect calculation. There is also the niggly issue of an errant validation error being reported.

This problem goes beyond advancements, though. The same issue will arise with injuries or any other permanent adjustment of the "trtBonus" field that occurs after character creation is completed. So we can't just fix this within the advancement mechanism. We need a more general solution that differentiates between bonuses applied during creation and post-creation.

The solution to this problem is actually pretty easy. We can define a new field to track only the bonuses applied during character creation. Anything that needs to apply a creation bonus must adjust this new field instead of "trtBonus". The new field should look like below.

```
<field
  id="trtCreate"
  name="Bonus During Creation"
  type="derived">
  </field>
```

To make use of this field as easy as possible, we'll define a new script macro to access the creation bonus, as shown below.

```
<scriptmacro
  name="traitcreation"
  param1="trait"
  result="hero.child[#trait].field[trtCreate].value"/>
```

With the macro in place, we can apply the appropriate changes to the various racial abilities. For example the Eval script for the "abSpirited" ability will change the code below.

```
#traitcreation[attrSpi] += 1
```

We can then define an Eval script on the "Traits" component that adds the creation bonus to the total bonus at an appropriate point in the evaluation cycle. This will ensure that all of the logic that keys on "trtBonus" remains perfectly valid. The best time to schedule this

script is probably at the very beginning of the Traits phase, yielding the script below.

```
<eval index="6" phase="Traits" priority="1"><![CDATA[
  field[trtBonus].value += field[trtCreate].value
  ]]></eval>
```

The final step is to modify the skill point calculation to use the new field instead of "trtBonus". This is a simple swap-out of one field for the other. The impacted line of code is shown below.

```
attrib += linkage[attribute].field[trtCreate].value
```

After a quick reload, our skill point calculations should be operating smoothly after advancement changes modify attributes.

INTEGRATED EDITOR SUPPORT (SAVAGE)

Our data files are basically complete. Our focus now shifts to the final steps necessary to prepare the data files for release to others. The first of these final steps is setting things up so that users can leverage the integrated Editor provided within HL.

Most gamers will have their own custom tweaks that they employ within their gaming group or that are unique to a particular game world. The goal is to enable them to easily add that material. You've invested quite a bit of time in creating your data files, so spending a small amount more to maximize the ability of others to use your files is a worthwhile investment.

USING THE EDITOR

We assume that you're at least moderately familiar with using the Editor for a different game system. If you haven't done so yet, take the time to review the documentation for one our licensed game systems and familiarize yourself with how the Editor works. After a little bit of experimenting, you should be able to appreciate how much simpler the Editor is to use than manually editing XML files. Our new objective is to wrap our data files up in a way that allows users to quickly and easily create new content via the Editor.

EDITOR BASICS

Internally, the Editor is driven by an assortment of "editthing" XML elements. Every "thing" is derived from a component set. Every type of thing that you want users to be able to create must have an "editthing" defined for it. Consequently, you'll need to define a separate "editthing" for every component set that users can create things for.

Within each "editthing", you must define one or more "inputthing" elements. Each "inputthing" corresponds to exactly one specific facet of a particular thing. For example, the value of a field is controlled via an "inputthing". So is whether a particular tag is assigned to the thing.

In an effort to keep everything as simple as possible for users, the Editor uses only a handful of simple mechanisms. These include checkboxes, picking an option from a list, entering a value, and a few others. Each "inputthing" specifies the exact user-interface mechanism to be used for customizing a single facet of the thing being edited. Along with the mechanism, it must also provide the internal details for how to map the visual mechanism to something concrete in the data files.

Let's look at an example. Skills can optionally require the user to specify a domain. Internally, this detail is controlled for skills via the "User.NeedDomain" tag. Within the Editor, we want the user to simply specify whether the skill requires the domain or not. The best way to handle an either-or situation like this is by showing the user a checkbox. So we would define an "inputthing" that uses the "tagcheck" mechanism. This mechanism presents a checkbox to the user and assigns a tag to the thing based on whether the checkbox is checked. If we specified the "User.NeedDomain" tag for the "tagcheck", that tag would be added to the thing if the checkbox is checked and omitted if unchecked.

The contents of an "editthing" are primarily just a list of "inputthing" elements. The order of the "inputthing" elements dictates the order in which everything is presented to the user within the Editor. This gives us complete control and allows us to incrementally get our data files to fully utilize the Editor. In the sections below, we'll be looking at this process in detail, with concrete examples. Through the process, you'll be exposed to most of the different "inputthing" mechanisms and how to use them.

USE TAGS WHENEVER PRACTICAL

Before we start implementing Editor support, we need to first address an important detail. The majority of the "inputthing" mechanisms center on the use of tags. The reason for this is that tags provide a critical benefit that simple values and text fields do not. Tags offer a restricted set of inputs that the user can select from.

In most situations, the data files for any game system will be designed around various small groupings of options. In Savage Worlds, all attributes and skills use a die type that ranges from "d4" to "d12". Hindrances can be either minor or major in severity, thereby having a cost of 1 or 2 points. The character's rank ranges through five levels, and edges can specify a dependency on the rank level.

The list goes on and on.

If we present the user with an empty field to enter a value into, we have no way of knowing that the user entered something valid. For example, if we present a field for the user to specify the minimum strength requirement for a ranged weapon, there is nothing to stop the user from entering a value of 42. Even if the user means well, the question arises whether he should be entering a value of 10 for a "d10" or a value of 5 to correspond with the technique we use internally. However, if we present a list of tags containing only the five valid die types, there is no opportunity for confusion and no opportunity for error.

There are still a variety of places where an arbitrary text or value field is exactly what we need. However, we will strive to use them only when they are truly appropriate. Keep this in mind as you're working on your own data files. We'll be running into a few situations below where we need to change our data files to leverage tags instead of fields.

TAKING INVENTORY

The final thing we should do before starting on our implementation is take inventory of our data files. We need to go through all the different files to identify which component sets we want to expose to users within the Editor. In general, this list should consist of any objects that we expect users will want to customize for their own games. Once we have the list culled out, we should then organize it into a reasonable sequence that will be easiest to implement.

The list we can identify for Savage Worlds is presented below, showing the component set id and the corresponding group of objects the user will be operating upon. With this list, we have a clear path for getting full Editor integration operational.

Attribute: Attributes
Skill: Skills
Trait: Derived Traits
RaceAbil: Racial Abilities
Edge: Edges
Hindrance: Hindrances
Reward: Hindrance Rewards
Equipment: Mundane Items
Melee: Hand Weapons
Ranged: Ranged Weapons
SpecWeap: Special Weapons
Armor: Armor
Shield: Shields
Vehicle: Vehicles
Arcane: Arcane Backgrounds
Power: Arcane Powers
Drawback: Arcane Drawbacks
Injury: Injuries
Race: Races
Creature: Creatures
Simple: Simple (used for validation rules and such)
Mechanic: Mechanics (used for auto-bootstrapped behaviors)

CHANGES NEEDED

Since we want to utilize tags whenever possible within the Editor, we need to go through each of the identified component sets above and look for places where we must make revisions. There are a total of four changes we need to make to the mechanics to streamline Editor integration.

The first change we'll notice is within the "Edge" component. The "edgIsWild" field is a simple "yes" or "no" situation. As such, it must be changed to the use of a tag, with the tag indicating a wildcard is required. To handle this, we'll define a "User.NeedWild" tag. We can then delete the field and go through all edges to assign the tag whenever the edge requires a wildcard. The final modification is within the "prereq" element on the "Edge" component, where the test of the field value must instead test the presence of the tag.

The second change is within the "MinRank" component, which will impact both edges and arcane powers. The minimum rank is specified via a field value, but it should be handled via a tag. We'll define a new "MinRank" tag group with tags corresponding to each rank level, as shown below.

```
<group
  id="MinRank">
  <value id="0" name="Novice"/>
  <value id="1" name="Seasoned"/>
  <value id="2" name="Veteran"/>
  <value id="3" name="Heroic"/>
  <value id="4" name="Legendary"/>
  </group>
```

With the tag group in place, we can delete the field and then tweak the "prereq" element on the component so that it checks for the tag instead of the field value (just like we did above). We can then go through all edges and arcane powers, converting all use of the "rnkMinRank" field over to the appropriate "MinRank" tag.

The third change is required within the "Hindrance" component. The "hinMajor" field has a simple "yes" or "no" behavior, so we can change it to key on the presence of a tag. We'll define a new "User.HindMajor" tag that indicates a hindrance starts out as a major one. Then we can go through all hindrances and assign the tag whenever a hindrance starts out as major.

This situation is a bit different from the wildcard field, though. The user can select the severity for some hindrances. Consequently, we cannot delete the field, since we need it to store the user-selected severity rating. What we'll do instead is configure the initial state of the field based on the presence of the tag. Since the field is user-modifiable, we can't do this in an Eval script, because we would end up clobbering the user-assigned state with our script every time. Instead, we'll handle it within a Creation script, since we can setup the initial state in the Creation script and then allow the user to adjust the field thereafter. Our script should look like below.

```
<creation><![CDATA[
  field[hinMajor].value = tagis[User.HindMajor]
  ]]></creation>
```

The final change we need to make is with weapons. The minimum strength requirement is currently handled via a field value. We need to change it in exactly the same way we handled the weapon die previously. If the tag is present, the corresponding requirement exists. The only caveat is that we must continue handling the presence of the weapon die as a strength requirement, with the separate requirement only applying when no weapon is given. Our new tag group should look like below.

```
<group
  id="StrReqDie"
  name="Strength Req Die">
  <value id="2" name="d4"/>
  <value id="3" name="d6"/>
  <value id="4" name="d8"/>
  <value id="5" name="d10"/>
  <value id="6" name="d12"/>
  </group>
```

Upon closer examination of the files, we'll find it difficult to actually delete the field. This means we'll need to use the tag to set the field value appropriately. Since it's not user-adjustable, we can change the field from "static" to "derived", then we can define an Eval script that maps the tag to the field value. The mapping is easy due to the proper use of the value possessed by each tag, resulting in the Eval script below.

```
<eval index="3" phase="Setup" priority="1000"><![CDATA[
  if (tagis[StrReqDie.?] <> 0) then
    field[wpStrReq].value = tagvalue[StrReqDie.?]
    endif
 ]]></eval>
```

The change impacts both the "WeapMelee" and "WeapRange" components, and all weapons based upon them. Any weapon that specifies the "wpStrReq" field must be modified to specify the corresponding tag instead. Beyond that, the only remaining change is within the "prereq" element on the "WeaponBase" component, which must change the tag instead of the field value on things.

At this point, we've got everything converted that must be converted. We can now begin define entries for Editor integration.

STARTING SIMPLE

All the pieces are in place, so it should be relatively straightforward for us to begin hooking everything into the Editor. We'll start with something simple, such as attributes. While few users will want to actually define a new attribute, there's no reason they can't, and making it possible is easy. All of our internal mechanisms just display a list of attributes, so adding a new one will merely increase the space needed to show all of them.

We must now take a look at the "Attribute" component set. There are three components that comprise the compset. The first is the "Attribute" component, which has nothing interesting that must be configured for a new attribute. The last is the "CanAdvance" component, which also has no behaviors that must be tailored for individual attributes. That leaves the "Trait" component, for which the only interesting field is the abbreviation ("trtAbbrev").

Now let's take a look at all of the attributes we've defined so far. There are six of them, including the one we added for super power skills. In assessing the attributes, there are two tags that we define on one or more of them. Each of these tags should be made customizable by the user when creating a new attribute. Combining that with the field identified above, we have the following three special pieces of information that we want the user to configure.

> **Abbreviation:** This will be a simple field where the user can enter the text he chooses. We can use a "field" input to capture the field value with the Editor.
>
> **Hide from User:** The super power attribute is hidden, so we must allow users to hide any new attribute that they may add. This state is controlled through the presence of a tag. We'll use the "tagcheck" input for Editor integration, since the tag will be assigned if the user checks the box.
>
> **Display Sequence:** The order in which attributes is shown depends on an explicit sequence defined by the author, which is dictated via a tag assignment. The user will need to assign a proper tag from a specific tag group, but the user can choose any tag that he wants to control the order. For this, we'll use a "tagpick" input for Editor integration, which presents the user with the option to select a tag from a specified tag group.

We can now define our "editthing". We'll specify the appropriate prefix to be used on all attributes ("attr"), along with a suitable description text and summary information. For each input, we'll provide a suitable name that tells the user what he's editing, as well as a information about how the input is used and what the implications are of any particular values or selections. Putting it all together yields an "editthing" that looks like the following.

```
<editthing
  compset="Attribute"
  name="Attribute"
  prefix="attr"
  description="Attributes are incredibly simple. They only have
an optional abbreviation."
  summary="Defines an Attribute to which starting attribute
points can be allocated by a character.">
  <inputthing
    name="Abbreviation"
    helptext="Specify a very short abbreviation for use in
displaying the attribute via its abbreviation (max 5 characters).">
    <it_field field="trtAbbrev"/>
  </inputthing>
  <inputthing
    name="Hide from User?"
    helptext="Specify whether this attribute is hidden from the
user and for internal use only.">
    <it_tagcheck group="Hide" tag="Attribute"/>
  </inputthing>
  <inputthing
    name="Display Sequence"
    helptext="Specify the sequence in which this attribute should
be shown to the user within lists of attributes. The unique id of
the tag must be an integer value.">
    <it_tagpick group="explicit" tag="?" require="yes"
deftag="1"/>
  </inputthing>
</editthing>
```

Editor support for attributes should now be in place, so reload the data files and let's test it. Launch the Editor via the "Tools" menu, then create a new data file. Click on the "Attribute" tab and then click on the "New (Copy)" button to create an attribute. Pick any of the standard five attributes and make a copy. Within the list of inputs below, you should see all three inputs we specified, with each having the proper behavior we expected.

Change the name to something conspicuous (e.g. "Zebra") and assign the thing a suitable unique id. Then specify a display sequence value of 10. Save the new attribute and click the button to test it now. Since we added an attribute, it's not user-selectable and doesn't appear automatically. We need to perform a quick-reload, after which it should appear everywhere it's supposed to. On the "Basics" tab, in the summary panel, on the character sheet, etc. Our attribute is working and we've got things integrated smoothly with the Editor.

THE NEXT FEW OBJECTS

Next on our list is skills. By looking through all the skills that we've defined, we can assess what mechanics we need to expose within the Editor. There are five items that we identify.

> **Abbreviation:** This is identical to attributes.
>
> **Linked Attribute:** Every skill must specify the attribute it is associated with. For linkages, the special "linkage" input is utilized, which will prompt the user appropriately based on the definition within the component.
>
> **Focus Requirement:** Some skills require a focus to be specified, which is controlled via a tag, so we'll use the "tagcheck" input.
>
> **Arcane Background:** Skills associated with arcane powers are tied to the proper arcane background. If a skill requires a background association, the user must be able to select it, so we'll use a "tagpick" input.
>
> **Show on TacCon:** Various skills are shown on the Tactical Console, and the user should be able to designate a skill for display. Such skills are identified via a tag, so we can use a "tagcheck" input for them.

This above analysis results in the following "editthing" definition.

```
<editthing
  compset="Skill"
  name="Skill"
  prefix="sk"
  description="Skills must specify a linked attribute that they are
based upon. Additional options can be specified, depending on
the behavior of the skill."
  summary="Defines a Skill to which starting skill points can be
allocated by a character.">
  <inputthing
```

```
      name="Abbreviation"
      helptext="Specify a very short abbreviation for use in
displaying the skill via its   abbreviation (max 5 characters).">
      <it_field field="trtAbbrev"/>
      </inputthing>
    <inputthing
      name="Linked Attribute for Skill"
      helptext="Select the attribute used to calculate the net attack
for this skill.">
      <it_linkage compset="Attribute" linkage="attribute"/>
      </inputthing>
    <inputthing
      name="Focus Required?"
      helptext="Specify whether this skill requires the user to
specify a focus (e.g. Knowledge).">
      <it_tagcheck group="User" tag="NeedDomain"/>
      </inputthing>
    <inputthing
      name="Arcane Background"
      helptext="Specify the arcane background that this skill is
associated with (for arcane skills only).">
      <it_tagpick group="Arcane" tag="?"/>
      </inputthing>
    <inputthing
      name="Non-Combat on TacCon?"
      helptext="Specify whether this skill is shown on the Tactical
Console when out of combat.">
      <it_tagcheck group="DashTacCon" tag="NonCombat"/>
      </inputthing>
    </editthing>
```

After skills, we have derived traits, for which we'll identify six facets that need to be customizable via the Editor. Mapping these to the corresponding "inputthing" elements is easy.

Abbreviation: Same as above.

Base Value: Traits like Parry and Toughness start with a base value of two, so we need a "field" input for this value.

Dashboard and TacCon Visibility: There are three separate areas where the trait can be listed, and each is governed by a separate tag, so we'll setup three different "tagcheck" inputs.

Display Sequence: Same as for attributes.

We continue onward with racial abilities. These are very different from the traits we've been handling thus far, but they are extremely simple and have only five facets that require customization. All of them are handled via use of a "tagcheck" input.

User Activation: Some abilities have in-play effects that the user can turn on or off, and they must identified via the "User.Activation" tag.

Selectable by Creatures: Many abilities can be selected for use when customizing creatures, and they are identified via a tag.

User Text: Some user-added abilities require the user to specify text, which is dictated by a tag.

User Value: Same as for text above, except a different tag is used.

Printable: A few abilities must be omitted from inclusion in character sheet output, and they are identified via a tag.

EDGES

As with the objects covered above, edges are actually rather simple to define for Editor integration. However, there are many facets to edges that need to be addressed. In the interest of providing one last complete example, we'll fully assess edges here.

Type of Edge: Every edge has a type, which must be selected from the list of possible tags via a "tagpick" input.

Minimum Rank: Every edge has a minimum rank requirement, which is dictated by a tag selection using "tagpick".

Wild Card Requirement: If an edge is only valid for wild cards, it can be assigned a tag via a "tagcheck" input.

User Activation: If an edge has in-play effects that the user can turn on or off, it is flagged via a tag using a "tagcheck" input.

Creation Only: Some edges are only selected at character creation and are identified via a tag using a "tagcheck" input.

Arcane Background: If an edge confers an arcane background, it must specify the proper tag via a "tagpick" input.

Printable: If an edge should be omitted from character output, it is designated with a tag via a "tagcheck" input.

Auto-Naming: Edges automatically have their name customized if user-selected facets exist, but it can be turned off via a tag controlled by a "tagcheck" input.

Domain Requirement: If an edge requires a domain from the user, a "tagcheck" input allows the tag to be assigned.

Domain Term: If an edge requires a domain, the term shown is controlled via a tag using a "tagpick" input. Since the user can specify any term they want, the input is marked as "dynamic", allowing the user to create a new tag with a custom name.

Menu Sources: If an edge requires the user to select customizations via a menu, the source from which to pull the menu options is specified via a tag using a "tagpick" input.

Menu Tag Expressions: Edges using menu selection must specify a tag expression to dictate the items shown in the menu, which can only be handled via a "field" input.

Distilling all of the above into an "editthing" results in the following.

```
 <editthing
    compset="Edge"
    name="Edge"
    prefix="edg"
    description="Edges have lots of facets, from requirements to
user activation to user   customization.{br}The customization
options cannot be combined - you can use menus or a domain."
    summary="Defines an edge that can be selected for a
character by the user.">
    <inputthing
      name="Type of Edge"
      helptext="Specify the category this edge gets classified in.">
      <it_tagpick group="EdgeType" tag="?" require="yes"
deftag="Background"/>
      </inputthing>
    <inputthing
      name="Minimum Rank Required"
      helptext="Specify the minimum rank a character must
possess to take this edge.">
      <it_tagpick group="MinRank" tag="?" require="yes"
deftag="0"/>
      </inputthing>
    <inputthing
      name="Wild Card Required?"
      helptext="Specify whether this edge is only selectable for
wild card characters.">
      <it_tagcheck group="User" tag="NeedWild"/>
      </inputthing>
```

```
    <inputthing
      name="Activated by User?"
      helptext="Specify whether this edge has an effect that can
be activated by the user via the In-Play tab.">
      <it_tagcheck group="User" tag="Activation"/>
      </inputthing>
    <inputthing
      name="Creation Only?"
      helptext="Specify whether this edge can only be selected for
a character during creation.">
      <it_tagcheck group="User" tag="CreateOnly"/>
      </inputthing>
    <inputthing
      name="Arcane Background"
      helptext="Specify any arcane background conferred by the
edge.">
      <it_tagpick group="Arcane" tag="?"/>
      </inputthing>
    <inputthing
      name="Omit from Printouts?"
      helptext="Specify whether this edge should be omitted from
character sheet output.">
      <it_tagcheck group="Print" tag="NoPrint"/>
      </inputthing>
    <inputthing
      name="Disable Auto-Naming?"
      helptext="Specify whether the automatic synthesis of names
for this edge should be disabled.">
      <it_tagcheck group="User" tag="NoAutoName"/>
      </inputthing>
    <inputthing
      name="Require Domain Specification?"
      helptext="Specify whether the edge requires the user to
specify a domain.">
      <it_tagcheck group="User" tag="NeedDomain"/>
      </inputthing>
    <inputthing
      name="Domain Term"
      helptext="Specify the term to use as a label for the domain. If
empty, the term 'Domain' is  used.">
      <it_tagpick group="DomainTerm" tag="?" dynamic="yes"/>
      </inputthing>
    <inputthing
      name="Menu #1 Source"
      helptext="Specify the source from which to retrieve choices
for display in the first menu.">
      <it_tagpick group="ChooseSrc1" tag="?"/>
      </inputthing>
    <inputthing
      name="Menu #1 Tag Expression"
      helptext="Specify the tag expression used to identify choices
for display in the first menu.   If empty, no first menu is shown.">
      <it_field field="usrCandid1"/>
      </inputthing>
    <inputthing
      name="Menu #2 Source"
      helptext="Specify the source from which to retrieve choices
for display in the second menu.">
      <it_tagpick group="ChooseSrc2" tag="?"/>
      </inputthing>
    <inputthing
      name="Menu #2 Tag Expression"
      helptext="Specify the tag expression used to identify choices
for display in the second menu.   If empty, no second menu is
shown.">
      <it_field field="usrCandid2"/>
      </inputthing>
    </editthing>
```

OTHER INTERESTING FACETS

The remaining objects can be defined by following the same logic as above and using the same mechanisms, so we won't be covering them here. There are are a few special situations, though, that we'll identify and discuss briefly.

There are a variety of component sets that derived from the "Gear" component. For things that are gear, there are two different cost behaviors that can be specified. There is both a "grCost" field and a "grLotCost" field. Users must have the distinction clearly explained to them within the Editor, else confusion will likely ensue. The difference is that the "grCost" field allows you to specify the cost for an individual item, while the "grLotCost" field allows you to specify the cost for a full "lot" of the item, as dictated by the "lotsize" attribute. Some game systems will list the cost for individual items, while others list the cost per lot-size purchases, and this allows users to enter the cost information either way.

Another facet of gear is that some items can be designated as "holders" of other gear. By default, all gear can be held within a backpack or other holder. However, users can also specify that a piece of gear can hold other items. There are two additional options. One allows a piece of gear to both be held within other holders and to hold items itself, as would be appropriate for a backpack or sack. The second option is for gear that can hold other items but cannot be held itself, as would be appropriate for a vehicle. Gear must allow the user to specify the holder behavior via a "tagpick" input that selects the appropriate tag.

Both weapons and vehicles possess an arbitrary assortment of special traits, such as "heavy weapon", "double tap", or "four-wheel drive". All of these traits are dictated via tags belonging to the same group, and any number of these traits can be assigned to a given thing. Handling this requires the use of the "taglist" input, which allows the user to select as many tags as he wishes from the specified tag group.

The final special behavior involves arcane backgrounds. There are a number of situations where an "Arcane" tag must be specified within the Editor, and these situations should generally require the user to select a tag that already exists. The exception is the "Arcane Background" thing, which should allow the user to define a new tag corresponding to the arcane background. This is achieved by setting the "dynamic" attribute to "yes" for the input, which tells the Editor to allow the user to add new tags.

STOCK PORTFOLIOS (SAVAGE)

It's invaluable to GMs to have an assortment of ready-made NPCs and villains to populate an unplanned encounter. The rulebooks for most game systems define a variety of such characters for exactly this purpose. You can create one or more portfolios of these characters and distribute them with your data files, providing GMs instant access to the character within HL. Portfolios containing these "stock" characters are accordingly referred to as "stock" portfolios.

BEHAVIORAL DIFFERENCES

Stock portfolios are standard portfolios. They contain one or more characters that the user is expected to import into his current portfolio. The key difference between stock portfolios and user-created portfolios is their placement and filename. Stock portfolios must reside in the data folder along with all the data files for the game system. They must also possess a ".stock" file extension.

This distinction accomplishes two goals. First of all, the files are kept separate from normal user-created portfolios, both based on filename and location. Secondly, the files can be readily identified

and included when packaging up your data files for sharing with others.

You can use HL to create a stock portfolio. The default file extension used for portfolios is ".por", but you can simply specify a file extension of ".stock" when saving the file. After that, you can open it and manipulate it normally within HL.

STOCK NPCS

The core rulebook for Savage Worlds only defines two stock NPCs for use within games. As such, there are only these two characters that we can add to a stock portfolio. These are the basic soldier and experienced soldiers.

Since the Savage Worlds rulebook covers some disparate time periods, the gear assigned to these stock NPCs needs to vary across time periods. So we'll create two separate stock portfolios, one for each of the major time periods (medieval and modern). Within each portfolio, we'll create two NPCs corresponding to the two soldier entries in the rulebook. We'll outfit each of the soldiers with pretty basic equipment for the time period, giving the experienced soldiers better equipment than the basic soldiers.

STOCK CREATURES

In many game systems, the creation of creatures requires a bit of effort. Consequently, it is usually advantageous to create a stock portfolio containing a wide assortment of monsters that GMs can quickly drop into encounters. Savage Worlds is a bit different, though.

Based on the way we setup creatures, a GM can add any creature he wants with a few mouse clicks. He can create a new character, make it a creature, then go to the chooser at the top and pick the desired creature. The total number of mouse clicks is about the same as importing creatures from a stock portfolio. As a result, there's no real benefit to having a stock portfolio for creatures, so we won't bother creating one.

SAMPLE CHARACTERS

In the rulebook for most game systems, at least one sample character is created that illustrates the overall character creation process. These characters are ideal for inclusion as sample portfolios with your data files. You can readily include these portfolios with your data files when distributing them (cover a little bit later).

Unfortunately, the Savage Worlds rulebook elects not to provide a detailed sample character. Since it's a good idea to always include at least one sample portfolio with your data files, we're left to concoct one of our own. For maximum benefit as an example, the character should show off a variety of facets of character creation, so we'll make sure to give him an arcane background and at least one hindrance.

We'll create our character in a folder beneath the "portfolios" folder used by HL. We'll name the folder the same as the one used for the data files (i.e. "savage"). This way, our sample character will appear in a suitable location when the user imports the data files into HL.

DATA FILE RELEASE (SAVAGE)

The data files that we've been developing are finally complete and soon ready to be shared with others. The sections below address a variety of final tasks associated with preparing our data files for distribution.

VERSION REQUIREMENT

Every new release of HL typically introduces new functionality within the Kit. If you utilize this functionality in your data files, you'll want to be sure that users don't try to load your files with an older version of HL. If they do, they will get errors when compiling the data files and think that the files are the problem. You can avoid this by designating the minimum version of HL that is required with your data files.

Within the definition file, the "release" element possesses a "required" attribute. This attribute can specify the minimum version of HL that your data files require. If you specify this correctly, any attempt by the user to load your data files into an older copy of HL will result in an appropriate error being reported. The user will be we told he needs a newer version of HL and pointed to the Updates mechanism to retrieve the update.

Since our data files are utilizing functionality introduced in V3.1 of HL, we need to specify that as our requirement. So the attribute is must be assigned the value "3.1".

VERSION NUMBER

When we release new updates to our data files, we're going to want to differentiate the newer version from any older versions. To accomplish this, each new release of our data files must be assigned a distinct version number.

The data file version number is used by HL to determine whether a new version is available. It is also used to detect when a loaded portfolio requires a newer version of the data files. This can happen if the portfolio is created on one computer and then loaded on a different one with a different version of the data files.

Just like HL itself, the version number used by data files has two pieces. There are both a major and minor version. In general, increasing the major version indicates that major changes and/or enhancements have been made to the data files. If the revision are relatively small in a new release, then the minor version number should typically be increased instead. When you increase the major version number, you should normally reset the minor version number to zero. Every time you release a new update to your data files, you should always increase the version number.

If you are releasing preliminary data files to friends for testing before you release the files widely, then you should probably assign a major version of zero and increment the minor version with each such release. When you officially release your data files for the very first time, you should normally assign a version of "1.0" (i.e. a major version of one and a minor version of zero). Since we're now ready to release our data files for the first time, that's the version we'll assign to our data files.

RELEASE NOTES

Within the definition file, you can specify a block of text referred to as "release notes". This block of text is shown to the user when they load your data files for the first time. Within the release notes, you should provide basic information to the user regarding your data files, including where to find further information (like the User Manual), legal text, and the like.

You can specify whatever information you like within the release notes. However, we recommend that you use a similar approach to how the data files for other game systems are handled. This ensures that users have a consistent experience across all game systems, which makes everything easier to use.

FAQ

You should ideally maintain a FAQ for your data files. The acronym "FAQ" is short for "Frequently Asked Questions". You can define your FAQ with two different methods, and either one is perfectly acceptable. The first option is to include the FAQ as a section within the User Manual for your data files (see below). The second option is to use the built-in FAQ mechanism provided by the Kit. This latter approach allows you to define the individual FAQ entries as part of the data files and then allow the Kit to synthesize the appropriate FAQ document as an HTML file.

As implied by its name, the FAQ contains an assortment of answers to questions that users will be asking about your data files. When you first develop your data files, the list will likely be short, but you will usually be able to anticipate at least a few questions that users will ask. If you provide answers before users ask them, it will make using your data files much easier for users and eliminate the annoyance of having to answer the questions when users bug you with them.

User Manual

The odds are that you'll want other gamers to utilize your data files. You've put lots of hard work into creating them, and sharing them with others will make lots of gamers very happy. However, these others won't know how to use your data files when they first load them. You're going to need to write a User Manual to explain that process.

Fortunately, the Skeleton Files provide a basic shell of a User Manual. This will be found as the file "manual.htm" in the "docs" folder beneath your data files. You can use this file as a starting point to evolve your own User Manual, as appropriate to the game system. Since this file is an HTML file, it can be viewed by virtually anyone on any computer. However, this also assumes that you can properly edit an HTML file to add your custom content.

If you are not familiar with the creation of an HTML document, you'll find it to be pretty simple. The overall structure is very similar to the XML format used by all the data files. In addition, all you need to do is plug in content. The structure can be left unchanged. This should result in a relatively straightforward process of evolving the User Manual with content suitable to your data files.

Building an Import File

At this point, your data files should be complete and ready to distribute. However, there are lots of files involved. Sharing all these files is a recipe for problems. Explaining to users how and where to place the data files on their computer is similarly going to result in problems for less technical users. Fortunately, HL includes a tool to make this process easy.

The tool is called "HLExport" and can be found on the Start menu with Hero Lab. This tool will take a complete set of data files and compress it into a single file with a ".hl" extension. This one file will contain all the information necessary for HL to automatically install everything into the correct place on a user's computer.

When the tool launches, you will be prompted for four pieces of information. First is the game system to be exported, which is identified by selecting the definition file for the game system. If you want to customize the set of files to be included, you can toggle individual files as you see fit. However, it is generally best to keep only the files you want to export in the "data" folder for your game system, with all other files being kept elsewhere. That way, the entire process is quick and easy.

The second piece of information is any sample portfolios that you want to distribute with your data files. It is generally a good idea to includes at least one sample character for the game system. If the rulebook defines assorted sample characters, those would be perfect for inclusion.

The third piece is the export file to be created. This file can be named anything, but it should typically be named appropriately for the game system (e.g. "savage.hl"). You may include the version number information within the filename (e.g. "savage.1.0.hl") if you want, but this is not generally done. The export file will contain everything and is the one file that can be easily shared with others. Users can readily import the contents of the file by going to the "Tools" menu and selecting "Import File", or by clicking on the "Import File" button that appears on the "Select Game System" form.

The fourth and final piece is notes/comments about the data files. These notes can be anything you want, but they should generally provide summary details about the data files being distributed. As a convenience, you can include this summary text within the definition file for the game system. This is done via the "summary" attribute within the "release" element. Whatever is defined for the "summary" attribute is automatically used as the default contents of the comments section.

Once you have everything entered the way you want, simply click on the "Generate" button and wait a moment. The HLExport tool will synthesize the appropriate ".hl" file. Once created, you can verify the file by attempting to import it into HL.

WARNING! Be careful not to import the file over the data files you're developing. By default, the data files will be placed in the same folder you pull them from. If there were any problems generating the ".hl" file, proceeding with the import in the default location could result in lost material. If you select an alternate installation folder, you'll be able to test everything without any difficulties.

Publishing Your Work

Once you have a completed ".hl" file, you can share your creation with others by simply giving them the ".hl" file. This file can be posted on websites for download, attached to emails, or whatever other mechanism seems appropriate to you.

To simplify the distribution process, Lone Wolf Development has a system already in place that allows authors to easily share their data files with others. Authors can post a notification about their data file updates on the Lone Wolf server. Once posted, all HL users will be able to see and access the data files directly from within HL.

Access to this mechanism is free. Once your data files are complete, simply contact technical support (mailto:support@wolflair.com) for details on how to distribute your data files. We'll set you up with an authoring account and you can then manage the notifications independently, releasing new updates at any time and without any involvement by Lone Wolf staff.

Changes in V3.2 (Savage)

The following changes were made to the Savage Worlds data files in the V3.2 release of Hero Lab.

1. Widened the space for the defensive value in the summary panel for armor and shields.

2. Fixed bug where character sheet output that continued onto subsequent pages did not properly position the right-side column of output.
3. Added Minimum Rank specification to Editor for Arcane Powers.
4. Added Forbidden Arcane Backgrounds specification to Editor for Arcane Powers.
5. Fixed bug where Horse and Warhorse were treated as normal gear holders and their weight would be applied to the character.
6. Eliminated options for menu label text from Editor, since no labels are ever shown in Savage Worlds.
7. Added stock portfolio containing all the various animals from the Bestiary.
8. Added stock portfolio containing all monstrous creatures from the Bestiary.
9. Errors with arcane powers now highlight the Arcane tab in red.
10. Unspent resources during character creation report validation warnings and highlight the appropriate tabs.
11. Added an in-play adjustment for Armor to enable easy application of the effects of the Armor arcane power.
12. Added Encumbrance and Load Limit details to the character sheet.
13. Added armor protection ratings on a per-location basis to the In-Play tab.
14. Fixed bug where ammunition was not having its quantity setup properly for tracking on the In-Play tab.
15. Eliminated the edit portal length from various portals that exceeded the maximum allowed length of the underlying field.
16. Fixed problem with the handling of abilities, where they were not being properly flagged on the actor for indication to the user.
17. Added support for entering "Ammunition" via the Editor.
18. All ammunition is automatically designated as a "GearType" of "Ammo".
19. Eliminated reliance on the "Wingdings 2" font by replacing all uses of that font with suitable alternates from the "Wingdings" font. The "Wingdings 2" font is not always available on all computers.
20. Changed the timing of the Eval scripts on the "Rich" and "Filthy Rich" edges to occur during the Setup phase.