GCA5 UPDATED DATA FILE INFORMATION

VERSION 3 WITH EXTENSIONS

Last Updated: December 20, 2022

This document is intended for the creation of files using the .GDF extension. This document covers the 3rd version of the GDF specification as well as extensions and enhancements supported by **GURPS®** Character Assistant 5*.*

CONTENTS

PURPOSE

This document will attempt to specify for you, as clearly as possible, the correct format and syntax for creating data files for use with ***GURPS*** *Character Assistant 5*. It may seem counter-intuitive, but this version of the GDF is version 3, not 4 or 5. There is no relationship between the GDF versions and the ***GURPS*** versions.

Be aware that the features available in the GDF format have grown through accretion over time: new features added as needed, rather than everything planned from the beginning. This means that some features that seem similar to other features may actually be structured quite differently, simply because they were implemented years apart, with a different history behind them.

Conventions

Examples of book contents or worked code look like this, even when filled with nonsense:

```
[Example]
This is an example.
```

And discussion of specific elements of the example will often reference those elements in a similar fashion, such as referring to [Example] in the block above.

Function or code templates look like this, and we generally try to avoid nonsense:

```
KEYWORD VARIABLE , CRITERIA KEYWORD [ OPTIONAL { SELECTION | SEGMENTS } ]
```

These templates often include extraneous spaces to clarify symbols used.

Most keywords, including tag names, are written INCAMELCASE to improve readability, even though GCA considers all tags to be lower case.

In the various code templates, such as the KEYWORD VARIABLE example above, punction and symbols are usually not color coded, even when they're required (such as the :: for referencing a tag). In some cases, they may be marked as keywords, usually in explanatory text.

File Structure

The data file is divided into many sections: header, author, sections for various traits, groups, lists, modifiers, and various specialty sections. Each of these sections has a particular structure. Aside from the Header section, which must be the very first section (and lines) in the data file, the order of the sections within the data file is not important.

Also, except for the Header section, the start of each section is marked by the name of the section enclosed by square brackets, like this:

```
[Advantages]
```

When GCA sees a line in the data file that looks like this, it knows that it is starting a chunk of that kind of data. You do not need to include this header for any section in your data file that doesn't include data.

Each section is explained in more detail below. If any of the explanation given below doesn't quite seem to make sense, you are encouraged to examine previously created data files to see how other people are doing things.

Tags

Tags are the basic structural item for most of the data sections in a GCA data file. The basic format of a tag is this:

```
TAGNAME(TAG INFO)
```

The TAGNAME specifies what information is being listed, and the TAG INFO contains the information that is being specified. The parenthesis before and after the TAG INFO are required. The opening parenthesis must come immediately after the TAGNAME and should never be separated from it by any whitespace. Again, for importance: do not include a space between the TAGNAME and the (, or GCA will not recognize the tag!

The tags that make up a trait need not be in any particular order. However, many of the trait definitions for a particular section of the data file include one or two pieces of data at the beginning of the definition that are not in tag form—those must be provided before any tags. For example, the Advantages section specifies that the name and cost of the advantage come first, then any desired tags, like so:

```
advantage item name (name extension), cost, tag1(tag1 info), tag2(tag2 info)
```

It is therefore required to provide the name and cost first, before providing the other tags you may wish to include.

General Structure Diagram

Here is a simple example diagram of what the general structure of a data file looks like. Remember that the Header must come first. Author usually follows Header for convenience, but otherwise order of the sections is not important. You do not need to include any sections for which you are creating no data.

```
Encoding=UTF-8
Version=3
* Other header lines

[Author]
* Copyrights and other Information about the file

[SkillTypes]
* Define various skill types and costs, by line

[BasicDamage]
* Basic sw and thr damage, by line

[ConvertDice]
* Definitions for converting bonuses to dice, by line

[Body]
<Body type>
* Body part definitions, by line

[Attributes]
* Attribute trait definitions, by line

[Advantages]
<Advantage category>
* Advantage trait defintions, by line

[Perks]
<Perk category>
* Perk trait defintions, by line

[Disadvantages]
<Disadvantage category>
* Disadvantage trait defintions, by line

[Quirks]
<Quirk category>
* Quirk trait defintions, by line

[Skills]
<Skill category>
* Skill trait defintions, by line

[Spells]
<Spell category>
* Spell trait defintions, by line
```

```
[Templates]
<Template category>
* Template trait defintions, by line

[Equipment]
<Equipment category>
* Equipment item defintions, by line

[Modifiers]
<Modifier group>
* Modifier item defintions, by line

[Groups]
<Group name>
* Group items, by line

[Lists]
<List name>
* List items, by line
```

General Information

Before we get into the overall structure of the data files, we should cover a few very general things about data files.

First, data files are plain text files, where each line of the file is one piece of data. For readability, you'll often break data items across multiple lines, but doing so requires the use of line continuation characters (see below).

Because data files are plain text, you should not use a word processor to save them, but rather a text editor, which is less likely to include strange characters or formatting. This is important, because many of the special characters that GCA relies on, such as quote marks, are often replaced by word processors with other characters, that GCA will not be able to recognize.

Second, blank lines are usually ignored by GCA, except when reading the Author section of a data file.

Lastly, there are some characters which have special meaning to GCA throughout a data file: comment markers and line continuation characters.

Special Characters

Comment Markers

*	An asterisk at the beginning of a line denotes a comment, and the whole line will be ignored (except in the Author section). An asterisk in the middle of the line will be taken to be the multiplication symbol, not a comment.
//	A pair of slashes together at any point in a line, when preceded by a blank space, denotes a comment. A pair of slashes together at the beginning of a line denotes a comment.

Anything following the comment marker will be ignored (except in the Author section).

Comments have precedence over all other special characters, so anything following a comment will never be seen. Never put line continuation characters after a comment marker, because GCA will never see them.

Line Continuation characters

_	An underscore character at the end of a line denotes a line continuation. The following line will be taken and appended at the end of the current line, in place of (and removing) the underscore character. Note that an underscore in a comment is ignored, so you should make sure not to try continuing comments with this character.
,	Commas are used to separate tags, and values within many tags (explained below). If you end a line with a comma, GCA will include the following line as part of the current line, appending it to the current line immediately following the comma, and negating the need to use an underscore.
|	The OR symbol (also known as the pipe symbol) is used in needs( ) tags to separate one group of possible prerequisite items from another, and in a few special cases in other tags, such as initmods(

), to separate groups of other tags. Ending a line with this character works the same way as ending the line with a comma.

Be aware that GCA often needs spaces between values in order to parse various commands or parts of lines correctly. GCA trims spaces from lines as they are read in from the data file, so if you are using a line continuation character such as an underscore, you may need to include a space before the underscore, to ensure that there is a space before the text being appended from the next line.

Restricted Characters

Because GCA uses certain characters to mean certain things internally, you should avoid using certain characters.

These characters should be considered restricted:

( ) { } , "

You can often use these characters safely within most tags, but it is better to avoid them unless they are part of a tag being used in the specified manner. In particular, be careful when using (and), because having an uneven number of either one can cause GCA to read data incorrectly from your data file.

Quote marks are never allowed as part of a name or name extension, and should always be avoided in any data, except when used as instructed to enclose data containing other special characters. In most such cases, braces are also allowed.

Single quote marks are acceptable data characters.

Math Characters

Many tags in GCA are *math-enabled*, meaning that GCA will evaluate a given expression to find the desired value. Because of this, GCA must be able to see and recognize math characters for what they are. There are a number of characters that are recognized as math characters in such tags, and you should take special care. These characters are all recognized as math characters:

( ) + - / * = > < & | ^ \

If any of these characters appears within the name of an item that is being used in a math section, that item name must be enclosed within quote marks, including any prefix tag, if applicable. For example, if a weapon skill was to default from the skill Axe/Mace at -4, then the default( ) tag for that item should look like this:

```
default("SK:Axe/Mace"-4)
```

Notice that the prefix tag of SK: is included within the quotes along with the name of the skill, but the math expression appears outside of the quotes.

Prefix Tags

Prefix tags are special codes tacked onto the front of trait names in references to tell GCA exactly what type of item that trait is. There are often several accepted prefixes for each type, but the preferred prefixes for the various trait types are:

ST:	for attributes/statistics
LA:	for languages
CU:	for cultural familiarities
AD:	for advantages
PE:	for perks
FE:	for features
DI:	for disadvantages
QU:	for quirks
SK:	for skills
SP:	for spells
TE:	for templates/meta-traits
EQ:	for equipment
GR:	for a group
LI:	for a list

In most cases, within tags, prefixes are required. Using prefixes speeds up GCA's ability to find the correct trait. In many cases, not using the prefixes will still work, and GCA will find the correct trait, but this will not work in all instances, and it will always be slower.

Section Detail Information

This section will cover the information that you need to write each section of a GDF3 data file.

Trait Names

All of the trait sections (Attributes, Languages, Cultures, Advantages, Perks, Features, Quirks, Skills, Spells, Templates, and Equipment) start each definition line with the name of the trait. This name consists of two parts: the name, and the name extension. The name extension is enclosed in parentheses after the base name portion, and is generally used for specifying specialties or other more specific information about the name.

If any portion of the name or name extension includes a comma, you must include the whole name portion in quotes or braces. For example:

```
Will (Against Interrogation), basevalue(10), etc.
"Will, against Seduction", basevalue(10), etc.
{Will, against Mind Control}, basevalue(10), etc.
```

Quote marks (aka double quotes) are never allowed as part of a name or name extension. Single quotes are acceptable.

Do not include two or more parenthetical parts in the name, as this can confuse GCA. Include no more than one, as the name extension.

Header

The Header section is one of the smallest and easiest sections to do, but it is extremely important that you do it correctly. The smallest complete Header section looks like this:

```
Version=3
```

The line containing the word 'Version=' followed by the version number of the data file format being used, in this case 3, is all that is required for the Header section, but it's not all that can be included. The Version line must be the absolute first line of the data file (or second, see below), coming before everything else, including comments. Do not put comments before this line!

Later enhancements to the GDF3 format allow for one line to be used before the Version=3 line, and that is if the data file is encoded in UTF-8, to support international character sets. If so, your Header section should start like this, instead:

```
Encoding=UTF-8
Version=3
```

Following the Encoding and Version lines can be several other header lines, which may appear in any order. You may also mix in comments, if desired, once you're beyond the Encoding and Version lines.

```
Encoding=UTF-8
Version=3
```

```
* The version information above MUST be the first line of the data file, unless Encoding=UTF-8 is used; then it
must be second.
Program=4.0.408
Description=Updated 22Jan2012. This is Armin’s example file.
Taboo="GURPS Basic Set 3rd Ed. Revised.gdf"
Requires="GURPS Basic Set 4th Ed.--Characters.gdf"
WorldBook=Gaia Campaign
Incomplete=Yes
LoadAfter=All
```

Description

```
DESCRIPTION=DESCRIPTION TEXT
```

The DESCRIPTION line allows you to include a description of what’s included in the data file, which will be displayed to the user when they are loading data files into a data set.

Incomplete

```
INCOMPLETE=YES
```

The INCOMPLETE line allows you to notify the user that the information in the data file is known to be incomplete, so they may find that certain traits or features that they may be expecting may not yet be available.

LoadAfter

```
LOADAFTER=FILENAMES
```

LOADAFTER designates that the file should be loaded after any of the files listed, if they're being loaded. Any files listed should include the full name of the GDF file (without path info) and can consist of multiple file names separated by commas. File names may be enclosed in quotes or braces if they might contain a comma.

LOADAFTER is intended for specifying files that this file may modify or make use of, but that are not actually required.

If you specify LOADAFTER=ALL, then this file should be loaded last, after all other data files. This is best for data files with a lot of data that overwrites or customizes data from a variety of other files.

Program

```
PROGRAM=GCA VERSION
```

The PROGRAM line allows you to specify the revision of GCA which is required to support any special features used in the data file. If an earlier version of GCA tries to load the file, it will be able to notify the user that an update will be needed to make use of the data in the file.

Requires

```
REQUIRES=FILENAMES
```

REQUIRES designates files that should be loaded *before* this file in the load order. The files listed should include the full name of the GDF file (without path info) that must be loaded and can consist of multiple file names separated by commas. File names may be enclosed in quotes or braces if they might contain a comma.

REQUIRES is intended for specifying files that this file is dependent on to work correctly in the loading stages, not necessarily for files that should be loaded when things are being added to the character. If the file includes commands that change previously loaded data definitions, for example, REQUIRES would be a good bet.

Taboo

```
TABOO=FILENAMES
```

The TABOO line allows you to specify data files that are known to be incompatible with the data in this file. You may specify multiple data files after the equals sign, with the various data file names separated by commas. File names may be enclosed in quotes or braces if they might contain a comma.

Note that TABOO is intended for alternate baseline files that most certainly will clash, such as Lite vs Basic Set. WORLDBOOK is intended for most other cases of possible incompatibilities.

Worldbook

```
WORLDBOOK=SETTING
```

The WORLDBOOK line allows you to specify that this data file is for a particular setting, and therefore GCA can notify the user of possible conflicts if they try to load other files marked with a different setting. Users should be aware that certain assumptions and data in the file may be incompatible with any other world book data file that they may be loading. (In general, you should only be loading files marked within one setting in any particular data set.)

All files containing worldbook specific data can now be safely flagged with the same WORLDBOOK=SETTING flag, and they will not trip GCA's worldbook flag message, but having two different SETTING values will trip the flag.

Author

The Author section is simple: everything following the [AUTHOR] marker in the file, until it reaches a new block marker, or the end of the file, is considered Author information.

Information that should most definitely be included in your Author block includes: the name of the data file and the program that it’s for, the copyright notice for the source book(s) used to create the data file (**don’t forget this!**), what the data file includes, and contact information so people can get in touch with you if they have questions or problems.

A sample Author section might look like this one, taken from ***GURPS*** Low-Tech:

```
[AUTHOR]
*************************************************************************
*
* GURPS® Character Assistant Data File
* Created: October 16, 2010
* Updated: May 31, 2011
* Requires GCA v4 or later!
*
* This data file contains information taken from the appendix of the GURPS Low-Tech sourcebook.
* GURPS Low-Tech is Copyright © 2010 by Steve Jackson Games Incorporated.
* All rights reserved.
*
* Information in this file includes:
*       New Equipment.
*
**************************************************************************
*
* Eric B. Smith                         - Data File Coordinator, Build Your Own Armor
* Emily Smirle (Bruno) - Everything else
*
* If you find any errors or omissions in this file please contact the Data File Coordinator at:
* ericbsmith42@gmail.com
*
* Or drop a message in the GCA4 forum at:
* http://forums.sjgames.com
*
**************************************************************************
*
* GURPS is a registered trademark of Steve Jackson Games Incorporated,
* used with GCA by permission of Steve Jackson Games. All rights reserved.
*
* The GURPS game is copyright © 2005 and other years by Steve Jackson
* Games Incorporated. This product includes copyrighted material from the
* GURPS game, which is used by permission of Steve Jackson Games Incorporated.
* All rights reserved by Steve Jackson Games Incorporated.
*
**************************************************************************
```

Settings

The [SETTINGS] block of a Book file has long been able to set certain basic settings for operation. However, these settings changed the *overall* configuration of how GCA operated, and that model doesn't fit well with how GCA5 allows for each character to be operating with different libraries, and therefore potentially the intention of very different intended settings.

The model now works a bit differently and has expanded. Default settings are either set internally for normally expected GURPS operation, or they are set on the Default Character Options pane of the Options dialog (depending on the type of setting). These go into the initial engine configuration, as loaded before any library is loaded. Then, when a library is created, it is created with a set of options that match the default engine configuration.

Once the library loads data, that data may change the defaults as set by the Settings blocks in book files. Finally, when a character is created, it is initialized with the settings in the library, and some of those can be changed on the Current Character Options pane of the Options dialog.

This is another change that reaches deep into GCA, but the effects of a mistake are unlikely to cause exceptions; instead the proper setting may not be honored as expected.

What settings can be set in Settings has been expanded with additional options, a quickie list of which is here:

```
CampaignName = value
DefaultTL = value
BasePoints = value
DisadLimit = value
QuirkLimit = value
HasDisadLimit = yes/no
HasQuirkLimit = yes/no
RuleOf = value
GlobalRuleOf = yes/no
UseDiceAddsConversion = yes/no
ModMultPercents = yes/no
AllowNonIQOptSpecs = yes/no
ModMaxNegPercentLimit = yes/no
NoDefaultLevelDiscount = yes/no
AllowStackingFortify = yes/no
AllowStackingDeflect = yes/no
AllowHalfPointSpells = yes/no
ShowMalf = yes/no
ShowBreak = yes/no
ShowLC = yes/no
```

SkillTypes

The [SkillTypes] section allows you to define the types of skills available, or to redefine existing types. Skill types are extremely important because they determine the basic features for skills in GCA.

Here is a sample SkillTypes section, taken from the Basic Set data file.

```
[SKILLTYPES]
N/A, cost(0/1), base(0), defaultstat(0), relname()
N|A, cost(0/1), base(0), defaultstat(%default), relname(def), subzero(yes)

DX/E, cost(1/2/4/8), base(-1), defaultstat(ST:DX), relname(DX)
DX/A, cost(1/2/4/8), base(-2), defaultstat(ST:DX), relname(DX)
DX/H, cost(1/2/4/8), base(-3), defaultstat(ST:DX), relname(DX)
DX/VH, cost(1/2/4/8), base(-4), defaultstat(ST:DX), relname(DX)
DX/WC, cost(3/6/12/24), base(-4), defaultstat(ST:DX), relname(DX)
```

```
IQ/E, cost(1/2/4/8), base(-1), defaultstat(ST:IQ), relname(IQ)
IQ/A, cost(1/2/4/8), base(-2), defaultstat(ST:IQ), relname(IQ)
IQ/H, cost(1/2/4/8), base(-3), defaultstat(ST:IQ), relname(IQ)
IQ/VH, cost(1/2/4/8), base(-4), defaultstat(ST:IQ), relname(IQ)
IQ/WC, cost(3/6/12/24), base(-4), defaultstat(ST:IQ), relname(IQ)

HT/E, cost(1/2/4/8), base(-1), defaultstat(ST:HT), relname(HT)
HT/A, cost(1/2/4/8), base(-2), defaultstat(ST:HT), relname(HT)
HT/H, cost(1/2/4/8), base(-3), defaultstat(ST:HT), relname(HT)
HT/VH, cost(1/2/4/8), base(-4), defaultstat(ST:HT), relname(HT)
HT/WC, cost(3/6/12/24), base(-4), defaultstat(ST:HT), relname(HT)

Will/E, cost(1/2/4/8), base(-1), defaultstat(ST:Will), relname(Will)
Will/A, cost(1/2/4/8), base(-2), defaultstat(ST:Will), relname(Will)
Will/H, cost(1/2/4/8), base(-3), defaultstat(ST:Will), relname(Will)
Will/VH, cost(1/2/4/8), base(-4), defaultstat(ST:Will), relname(Will)
Will/WC, cost(3/6/12/24), base(-4), defaultstat(ST:Will), relname(Will)

Per/E, cost(1/2/4/8), base(-1), defaultstat(ST:Perception), relname(Per)
Per/A, cost(1/2/4/8), base(-2), defaultstat(ST:Perception), relname(Per)
Per/H, cost(1/2/4/8), base(-3), defaultstat(ST:Perception), relname(Per)
Per/VH, cost(1/2/4/8), base(-4), defaultstat(ST:Perception), relname(Per)
Per/WC, cost(3/6/12/24), base(-4), defaultstat(ST:Perception), relname(Per)

Tech/A, cost(1/2/3), base(0), defaultstat(%default), relname(def), subzero(yes)
Tech/H, cost(2/3/4), base(0), defaultstat(%default), relname(def), subzero(yes)
```

Like many other sections in the GDF, SkillTypes uses one line per item, in a tagged format, with the very first data item being the name of the skill type.

The name for each skill type is usually an attribute, a slash, and then the difficulty of the skill, such as HT/E for an Easy skill based off of HT. This is not required but is encouraged since that's how users will expect to see them. If you need to define a custom skill type, you are encouraged to follow this convention, although it's necessary to vary some. For example, techniques use Tech/A or Tech/H, and combinations use Combo/2 and Combo/3.

Tags

Since this section uses its own special tags, we'll cover them here.

base()

This is the amount subtracted from the base score of the related attribute or other starting score, to provide the value that is one step below what should be gained from the first increment. In other words, if DX/E gives you a skill level of DX-0 for one point, then the BASE() for DX/E should be -1, or base(-1), because -1 is one step below 0.

The default value is 0.

cost()

This is the cost per level (cost per step) of the skill. Costs should be separated by slashes. Specify as many costs as required. GCA will use the difference between the last two costs specified as the cost for any steps beyond the given progression.

The default value is 0.

defaultstat()

This is the attribute upon which the skill type is normally based. If a skill definition doesn't specify the attribute it's based on, this value will be used. You may specify an attribute name, a number, or the %DEFAULT keyword, which means that the skill type is used for techniques, and the base value will be based off of the skill's defaulted level.

The default value is DX.

relname()

This is the attribute name to display when showing relative levels, as the value being "stepped off" from. This allows you to specify a shorter version of a name than might otherwise be displayed if GCA had to display the full attribute name, such as Per instead of Perception.

stepadds()

This tag allows you to specify by how much each step up of the skill increases the skill level. This is specified in the same way as COST(), with values separated by slashes. For example, stepadds(1) would mean each step adds one to the current level, while stepadds(2) would add two for each step, and stepadds(1/3/6/10) would add 1 for the first level, 2 for the second, 3 for the third, and 4 for each increment thereafter.

The default value is 1.

subzero()

This tag allows you to specify that having a level below 0 is okay. Normally, GCA does not allow skill levels below 0. You may include subzero(true) or subzero(yes) to turn this on; any other value will result in FALSE.

The default value is FALSE.

zeropointsokay()

This tag allows you to specify that it is okay to have a level if there are no points spent on the skill. Normally, GCA requires spending the minimum specified cost to have a level in the skill. You may include zeropointsokay(true) or zeropointsokay (yes) to turn this on; any other value will result in FALSE.

Note that GCA interprets skill types with defaultstat(%default) to be techniques, which results in it ignoring this tag value for those skills.

The default value is FALSE.

BasicDamage

The [BASICDAMAGE] section allows you to define the basic damage per ST score used in GCA.

If you include this section, you must define every possible level, because GCA replaces any previously loaded values with the ones you define; you may not redefine only a portion of the data in this section.

Here is a sample BasicDamage section, taken from the Basic Set data file. (This sample is incomplete, having been trimmed down for this example.)

```
[BASICDAMAGE]
st(1), thr(1d-6), sw(1d-5)
st(2), thr(1d-6), sw(1d-5)
st(3), thr(1d-5), sw(1d-4)
st(4), thr(1d-5), sw(1d-4)
st(5), thr(1d-4), sw(1d-3)

* Other values clipped for this example. Please see the Basic Set data file
* for the full listing.

* The LAST item in the list is always the item that is to be used
* for anything that didn't fall under the preceding items.

st(0), thr((@int(ST:Striking ST/10)+1)d), sw((@int(ST:Striking ST/10)+3)d)

* you must use the extra set of parens to separate the math part
* from the 'd' for the dice.
```

Like many other sections in the GDF, BasicDamage uses one line per item, in a tagged format. There is no name section at the front.

It's important that you list the values in increasing ST order, as GCA references it in that fashion to find the correct values for any given ST value. GCA will check all ST() values until it finds the last one that is less than, or equal to, the value it's checking against, and then use the data given for that value.

It's also important that the last data item you specify include the formulas for finding any values beyond those provided in the preceding data.

Tags

Since this section uses its own special tags, we'll cover them here.

st()

This tag specifies the ST value that is less than, or equal to, the ST value being checked against, for which the THR() and SW() tags apply.

thr()

This tag specifies the basic thrust damage for the given ST value.

sw()

This tag specifies the basic swing damage for the given ST value.

ConvertDice

The [CONVERTDICE] section allows you to define the breakpoints GCA will use to convert damage bonuses into damage dice if that optional rule is being used (see MODIFYING DICE + ADDS on p. B269).

If you include this section, you must define every break point, because GCA replaces any previously loaded values with the ones you define; you may not redefine only a portion of the data in this section.

Here is a sample ConvertDice section, taken from the Basic Set data file.

```
[CONVERTDICE]
* break()'if the bonus is this Value or more ...
* adddice()   '... add this Value to the damage dice ...
* subtract()  '... then subtract this amount from the bonus
*
* you need to start with the biggest break you want to deal with,
* and work down to the smallest, because GCA doesn't sort these either
*
break(7), adddice(2), subtract(7)
break(4), adddice(1), subtract(4)
```

Like many other sections in the GDF, ConvertDice uses one line per item, in a tagged format. There is no name section at the front.

It's important that you list the values from largest break point to smallest, as GCA will use the first BREAK() value that it finds that will work, and will only look at subsequent break points if it's unable to use earlier ones.

GCA will work through the various breakpoints specified in the ConvertDice section until it's no longer able to apply any of them to a damage amount. Each time GCA applies a breakpoint, it will start over at the top again for the next pass.

Tags

Since this section uses its own special tags, we'll cover them here.

break()

This tag specifies what the value of the damage bonus should be to use this break point. If the damage bonus is this value or greater, GCA will use this breakpoint.

adddice()

When using this breakpoint, GCA will add this many dice to the damage dice.

subtract()

When using this breakpoint, GCA will subtract this amount from the damage bonus.

Body

The [BODY] section allows you to define body types for use with GCA. Body types are necessary for applying armor to your character.

Here is a sample Body section, taken from the Basic Set data file.

```
<Humanoid>
description(Stock humanoid. B 552)
name(Vitals), group(Skin, All, Full Suit, Body, Torso, Vitals)
name(Head), group(Head, Skin, All)
name(Eyes), group(Head, Eyes, All), display(-1), expanded(-1), posx(149), posy(53)
name(Left Eye), group(Head, Left Eye, Eyes, All)
name(Right Eye), group(Head, Right Eye, Eyes, All)
name(Neck), group(Neck, Body, Full Suit, Skin, All), display(-1), expanded(-1), posx(137), posy(114)
name(Skull), group(Head, Skull, Skin, All), basedr(2), dr(2), display(-1), expanded(-1), posx(339), posy(43)
name(Face), group(Head, Face, Skin, All), display(-1), expanded(-1), posx(309), posy(104)
name(Torso), group(Torso, Body, Full Suit, Skin, All), display(-1), expanded(-1), posx(220),
posy(193),width(100),height(50)
name(Groin), group(Groin, Body, Full Suit, Skin, All), display(-1), expanded(-1), posx(393), posy(474)
name(Arms), group(Arms, Limbs, Full Suit, Skin, All), display(-1), expanded(-1), posx(403), posy(246)
name(Left Arm), group(Left Arm, Arms, Limbs, Full Suit, Skin, All)
name(Right Arm), group(Right Arm, Arms, Limbs, Full Suit, Skin, All)
name(Hands), group(Hands, Full Suit, Skin, All), display(-1), expanded(-1), posx(420), posy(310)
name(Left Hand), group(Left Hand, Hands, Full Suit, Skin, All)
name(Right Hand), group(Right Hand, Hands, Full Suit, Skin, All)
name(Legs), group(Legs, Limbs, Full Suit, Skin, All), display(-1), expanded(-1), posx(351), posy(587)
name(Left Leg), group(Left Leg, Legs, Limbs, Full Suit, Skin, All)
name(Right Leg), group(Right Leg, Legs, Limbs, Full Suit, Skin, All)
name(Feet), group(Feet, Full Suit, Skin, All), display(-1), expanded(-1), posx(349), posy(662)
name(Left Foot), group(Left Foot, Feet, Full Suit, Skin, All)
name(Right Foot), group(Right Foot, Feet, Full Suit, Skin, All)
name(Body), group(Body, Full Suit, Skin, All)
name(Full Suit), group(Full Suit, Skin, All)
name(Skin), group(Skin, All)
name(All), group(All)
```

Each body type is specified using angle brackets, as you'd define a category for a trait section.

Once you've specified the body type, each part of the body is specified in the standard tagged format, one item per line. The body part definition does not use a name section at the front, but uses a NAME() tag.

Tags

Since this section uses its own special tags, we'll cover them here.

description()

This is a special tag that should only appear once per body, which allows you to include a description of what the body is, and anything else the user should know.

name()

This is the name of the part of the body being defined. These names should correspond to various areas that can be covered by armor. There's generally no point in defining body parts that don't ever have pieces of armor with corresponding LOCATION() values.

display()

This tag specifies whether the body part should have a UI element within GCA displayed to the user by default. Use -1 for TRUE or 0 for FALSE, such as display(-1). The default value is FALSE.

expanded()

This tag specifies whether the body part should have an expanded and open editing block in the body parts listing within GCA by default. Use -1 for TRUE or 0 for FALSE, such as expanded(-1). The default value is FALSE.

group()

This tag lists all the body parts with which this part should be grouped. This is important, as more comprehensive groups that include other body parts are specified through the use of the GROUP() tag. For example, the All body part includes every other body part, because every other body part lists it in their GROUP() tag.

Note that groups of parts need not have corresponding body parts specified in order to group parts together. In the example listing above, for example, the Limbs group includes Arms, Left Arm, Right Arm and other parts, even though there is no listing for Limbs itself.

posx(), pos(y)

If display(-1) is being used, POSX() and POSY() tell GCA the location (x and y coordinates) where the edit box should be displayed to the user on the "paper doll" armor diagram.

HitTables

This section allows you to specify hit location tables of various sorts, usually to match a body type.

Here is a sample hit table taken from the Basic Set data file.

```
<Quadruped>
description(A creature with four legs and no arms. B 553)
roll(-), location(Eye), penalty(-9)
roll(3-4), location(Skull), penalty(-7)
roll(5), location(Face), penalty(-5)
roll(6), location(Neck), penalty(-5)
roll(7-8), location(Foreleg), penalty(-2), notes(*)
roll(9-10), location(Torso), penalty(0)
roll(11), location(Torso), penalty(0)
roll(12), location(Groin), penalty(-3)
roll(13-14), location(Hind Leg), penalty(-2), notes(*)
roll(15-16), location(Foot), penalty(-4), notes(*)
roll(17-18), location(Tail), penalty(-3), notes(T)
roll(-), location(Vitals), penalty(-3)
note(*), text(If using random hit location, roll 1d: 1-3 is right, 4-6 is left. If it is somehow holding a shield,
double the penalty to hit: -4 for a limb, -8 for an extremity.)
note(T), text(Tail: If a tail counts as an Extra Arm or a Striker, or is a fish tail, treat it as a limb (arm, leg) for
crippling purposes; otherwise, treat it as an extremity (hand, foot). A crippled tail affects balance. For a ground
creature, this gives -1 DX. For a swimmer or flyer, this gives -2 DX and halves Move. If the creature has no
tail, or a very short one (like a rabbit), treat as "torso.")
```

Each new hit table is specified using angle brackets, as you'd define a category for a trait section. If you use a name that matches one of the BodyType entries, GCA will be able to automatically select a matching HitTable when the user changes body types in GCA.

Using a section with a HitTable name that already exists will replace the existing one with the new one.

Once you've specified the table, there are three sections, each differentiated only by the different sets of tags that apply, so you must use the correct tags, one item of each type of data per line.

Description

This section is just one line of one tag, and uses the DESCRIPTION() tag to describe the hit location table.

Table

This section sets up the table itself, such as what values are rolled to randomly hit a location, what that location is, and what the penalty for targeting it is.

Here are the tags used for that:

ROLL()	Enter a single value, such as 5; a range of values, such as 9-10; or a – to indicate that the location can't be hit randomly.
LOCATION()	Enter the name of the location for this roll, such as Face or Foreleg.
PENALTY()	Enter the penalty to hit this location, such as -5 or -2.
NOTES()	Enter the symbol that will match up to the Notes block data below. If there are multiple notes symbols here, separate them with commas.

Notes

This section specifies the notes for the lines that define the Table above.

NOTE()	This is the symbol for one specific notation. One symbol per line here, and each must be unique.
TEXT()	This is the text describing what the notation actually is.

Flag Symbols

AKA those little icons marking supernatural, exotic, etc.

This allows specifying images for the symbols representing various flags within GURPS (such as those for Supernatural, Physical, or Exotic advantages and disadvantages; see B32), as well as how to apply them.

```
[Symbols]
Mental Advantage, mental_16.png, Ads where cat listincludes Mental
Mental Disadvantage, mental_16.png, Disads where cat listincludes Mental
Physical Advantage, physical_16.png, Ads where cat listincludes Physical
Physical Disadvantage, physical_16.png, Disads where cat listincludes Physical
Social Advantage, social_16.png, Ads where cat listincludes Social
Social Disadvantage, social_16.png, Disads where cat listincludes Social
Exotic Advantage, exotic_16.png, Ads where cat listincludes Exotic
Exotic Disadvantage, exotic_16.png, Disads where cat listincludes Exotic
Supernatural Advantage, supernatural_16.png, Ads where cat listincludes Supernatural
Supernatural Disadvantage, supernatural_16.png, Disads where cat listincludes Supernatural
```

There are three sections per line, separated by commas:

```
NAMEOFSYMBOL , IMAGEFILE , CRITERIA
```

The first section is NAMEOFSYMBOL which is the name you give to the symbol This is so that every symbol can be unique internally, which will allow for using the same name in another file to replace it if defined earlier, such as with a new image or a new rule. The name should not include commas or other punctuation.

The second section is IMAGEFILE and is the file name for the image file on disk to use as the icon for the symbol. It should be in one of the image bins, so that GCA can find it. (Note that GCA will check the User image bin before the System image bin, so that users can change the default system images, if desired, simply by adding replacement images with the same name to their own image bin.)

The last section is the CRITERIA specification, which is the rule for applying the symbol. This is covered in Criteria below.

Note that the system allows for specifying whole new types of symbols, should users wish to make use of the system for custom tagging their traits. Custom images should be 16 pixels tall (because GCA will not resize them to fit, and 16 pixels is the standard height for other icons in trait lists) but may vary in width. The best image type for them is PNG, with alpha channel transparency if desired, but solid backgrounds are okay if you like them. BMP files may also be used.

*There is a pretty great set of symbols now available for GCA, thanks to Eric Smith, which you can get through the Package Manager, and then activate by loading the **GCA5 Symbols.GDF** book in a library. A huge variety of symbols are included for nearly all types of traits.*

Commands

The [SYMBOLS] block supports two commands at this time, which must be within the block to work:

#Clear

> #CLEAR

#CLEAR stands alone, and when encountered causes GCA to delete all currently defined symbols, so that any defined after that point will be the only ones used.

#Delete

> #DELETE LISTOFNAMES

#DELETE specifies a list of symbols to be deleted. This should be a comma separated list of the names of the symbols as defined in the data files.

> #Delete Mental Advantage, Mental Disadvantage

Criteria

The basic criteria selection specification is the same as you'd find in a #BUILDSELECTLIST but has been greatly expanded.

The basic criteria specification looks like this:

> TRAITTYPE WHERE TAG [IS | ISNOT | INCLUDES | EXCLUDES | LISTINCLUDES | LISTEXCLUDES] TAGVALUE

such as

> Ads where cat listincludes Mental

which is handy, but limited, allowing only for a single comparison statement after the WHERE keyword.

The criteria system here is more robust, and you can specify as many comparisons as you need to be specific, even using and/or symbols and grouping within parentheses if necessary. The TRAITTYPE declaration is fixed, however.

So, now you can expand everything after the WHERE keyword, so you could have that portion of the criteria be something like this nonsense example:

> TAG compare TAGVALUE | TAG compare TAGVALUE , TAG compare TAGVALUE , (TAG compare TAGVALUE | TAG compare TAGVALUE)

Basically, it's handled like Needs, where it's designed to break apart criteria on OR markers first, so if you want to do a simple OR selection among a list of AND items, enclose them in parens as you'd do in a Needs.

In the pseudo-example above, GCA would see the example as two major OR blocks, either of which could be True for the symbol to be applied. They are:

> TAG compare TAGVALUE

OR

> TAG compare TAGVALUE , TAG compare TAGVALUE , (TAG compare TAGVALUE | TAG compare TAGVALUE)

Notice that the second is much longer and includes a sub-clause OR in parens.

You can also use & or + as the AND symbols, instead of just commas, in case you might prefer that. Sometimes that might be more clear. Unfortunately, the | is the only symbol for OR, and you can NOT use the words AND or OR instead of the symbols. So, the above example could also be written like this:

> TAG compare TAGVALUE | TAG compare TAGVALUE & TAG compare TAGVALUE & (TAG compare TAGVALUE | TAG compare TAGVALUE)

In addition to the expanded structure in general, the comparisons that are allowed have been expanded as well, so you can now use any of the comparisons shown here:

> TAG { = | > | < | <> | >= | <= | IS | ISNOT | INCLUDES | EXCLUDES | LISTINCLUDES | LISTEXCLUDES } TAGVALUE

Remember the usual rules, and enclose TAGVALUE in quotes or braces if it includes commas or spaces.

Body Images

GCA now supports assigning custom images for body types.

Set these in data files using the [BODYIMAGES] block, where each section in the block corresponds to a body type, and each line in each section corresponds to an image file to use for that body type.

A BodyImages block might look like this:

```
[BodyImages]
<Humanoid Expanded>
image_body_locations.png
```

Valid image types are BMP and PNG and whatever else .Net supports natively.

GCA will search for the images in the following order:

1) using any fully qualified path that is specified;
2) using any specified path using the standard shortcut variables (%app%, %sys%, %user%, %userbase%);
3) in the User image bin (%user%\images\);
4) in the System image bin (%sys%\images\).

Attributes

The [ATTRIBUTES] section allows you to define new attributes, or to redefine attributes that have been defined in previously loaded files. (Note: It's considered better practice to use commands (see elsewhere) to replace certain tag values in existing traits, rather than redefining them.

Replacing tag values allows for replacing just the specific feature that you need changed, rather than having to include the full trait definition. This means you're less likely to end up with an outdated definition if the original source data is changed.)

To define a new attribute, you must specify the attribute name followed by any other tags needed to complete the trait definition. The only required part of the definition is the name of the attribute. Following the name should be a comma separated list of any other tags necessary to complete the trait.

To redefine an existing attribute, the only requirement is that the attribute name you specify be exactly the same as the name of the existing attribute you wish to overwrite.

Here is a small sample Attributes section that redefines Will to be based on 10 instead of IQ, and creates a new attribute called Honor.

```
[Attributes]
Will, basevalue(10), step(1), maxscore(1000000), minscore(0 - me::syslevels), up(5), down(-5), mainwin(6)
Honor, basevalue(ST:Will), step(1), maxscore(20), minscore(1), up(10), down(-10), display(no)
```

The Attributes section in data files may now specify categories for attributes, and GCA will now also allow using the <CATEGORY> format to denote categories of attributes, just like every other type of trait.

Advantages

The [ADVANTAGES] section allows you to define new advantages, or to redefine advantages that have been defined in previously loaded files. (Note: It's considered better practice to use commands (see elsewhere) to replace certain tag values in existing traits, rather than redefining them. Replacing tag values allows for replacing just the specific feature that you need changed, rather than having to include the full trait definition. This means you're less likely to end up with an outdated definition if the original source data is changed.)

To define a new advantage, you must specify the advantage name, followed by the cost notation, followed by any other tags needed to complete the trait definition. The only required parts of the definition are the name of the advantage and the cost notation. Following the required parts should be a comma separated list of any other tags necessary to complete the trait.

To redefine an existing advantage, the only requirement is that the advantage name you specify be exactly the same as the name of the existing advantage you wish to overwrite.

The cost notation should be the cost of the trait as specified in the source. If it is a leveled trait, the costs should be separated with a forward slash, such as 5/10. You may define as many levels of cost as needed, but GCA will use the difference between the last two costs as the cost increment for any further levels that don't have specified costs. You must specify at least two levels of cost for any leveled trait.

Here is a small sample Advantages section. Notice that line continuation characters are being used to make the Chameleon trait easier to read, and to make Dark Vision fit into the example area.

```
[Advantages]
```

```
<Exotic Physical>
Breath-Holding, 2/4, page(B41), cat(Exotic, Physical)
Chameleon, 5/10, mods(Chameleon), page(B41), cat(Exotic, Physical),
        conditional(_
                                +2 to SK:Stealth when "perfectly still, unless clothed",
                                +1 to SK:Stealth when "moving, unless clothed",
                                +1 to SK:Stealth when "perfectly still, and clothed"_
                                )
Dark Vision, 25, mods(Dark Vision), page(B47), cat(Exotic, Physical),
        taboo(AD:Night Vision, DI:Night Blindness, DI:Blindness)
```

You may break the Advantages section into different categories by enclosing the category name in angle brackets on its own line. In the sample above, a category called Exotic Physical is defined. All of the trait definitions that follow the category (until a new one is encountered) will include that category in their CAT() tags, in addition to any other categories that may already be in the CAT() tag. This allows users to look for traits by category, and allows you to define any categories you need, while not having to include the trait definition under each different category that applies in the data file.

You should not include a colon (:) in the category name, because that has a special meaning. You should also not include any additional angle brackets (< or >) or commas (,).

Languages

These are advantage-like traits, and the data is structured that way.

```
[LANGUAGES]
<Language>
Language, 2/4, page(B24), upto(3 LimitingTotal), mods(Language),
        levelnames([None], Broken, Accented, Native), cat(Mundane, Social, Language, Language Spoken,
        Language Written, Social Background), taboo(Native Languages > 1),
        x(#InputToTagReplace("Specify the language here:", name, , "Language"))
```

Cultural Familiarities

Other than the section marker being [CULTURES], Cultural Familiarities are defined like Advantages. See Advantages above. Note that Cultures should generally be specified with a cost of 1.

```
[CULTURES]
<Cultural Familiarity>
Cultural Familiarity, 1, page(B23), mods(Cultural Familiarity),
        cat(Mundane, Social, Cultural Familiarity, Social Background), taboo(Native Cultural Familiarities > 1),
        x(#InputToTagReplace("Specify the culture you're familiar with here:", name, , "Cultural Familiarity"))
```

Perks

Other than the section marker being [PERKS], Perks are defined like Advantages. See Advantages above. Note that Perks should generally be specified with a cost of 1.

Here is a small sample Perks section.

```
[Perks]
<Perks>
Accessory, 1, page(B100)
Alcohol Tolerance, 1, page(B100)
```

Features

These are advantage-like traits, and the data is structured that way. Features generally have a cost of 0.

```
[FEATURES]
<Features>
_New Feature, 0, noresync(yes),
        x(#InputToTagReplace("Please enter the name of this Feature:" , name, ,"New Feature")_
        )
```

Disadvantages

Other than the section marker being [DISADVANTAGES] and having negative costs such as -5/-10, Disadvantages are defined like Advantages. See Advantages above.

Here is a small sample Disadvantages section.

```
[Disadvantages]
<Exotic Physical>
Shadow Form, -20, mods(Shadow Form Disadvantage), page(B83), cat(Exotic, Physical), taboo(AD:Shadow
Form)
Cold-Blooded, -5/-10, upto(2), page(B127), cat(Exotic, Physical),
        levelnames(You "stiffen up" below 50°, You "stiffen up" below 65°),
        conditional(=+2 to ST:HT when "resisting effects of temperature")
```

Quirks

Other than the section marker being [QUIRKS] and having a negative cost, Quirks are defined like Advantages. See Advantages above. Note that Quirks should generally be specified with a cost of -1.

Here is a small sample Quirks section.

```
[QUIRKS]
<General>
_Unused Quirk 1, -1, page(B163)
_Unused Quirk 2, -1, page(B163)
```

Skills

The [SKILLS] section allows you to define new skills, or to redefine skills that have been defined in previously loaded files. (Note: It’s considered better practice to use commands (see elsewhere) to replace certain tag values in existing traits, rather than redefining them. Replacing tag values allows for replacing just the specific feature that you need changed, rather than having to include the full trait definition. This means you're less likely to end up with an outdated definition if the original source data is changed.)

To define a new skill, you must specify the skill name, followed by the skill type notation, followed by any other tags needed to complete the trait definition. The only required parts of the definition are the name of the skill and the skill type notation. Following the required parts should be a comma separated list of any other tags necessary to complete the trait.

To redefine an existing skill, the only requirement is that the skill name you specify be exactly the same as the name of the existing skill you wish to overwrite.

The skill type notation should be the type of the skill as specified in the [SKILLTYPES] section in the Basic Set data file, or elsewhere in your data file. These type notations are generally similar to IQ/A or DX/H (denoting an Average IQ-based skill or a Hard DX-based skill, respectively). The Basic Set data file defines all the standard types for normal ***GURPS*** skills, but other data files may introduce other types.

Here is a small sample Skills section.

```
[Skills]
<Animal>
Animal Handling (Raptors), IQ/A, default(IQ - 5), page(B175), cat(_General, Animal)
Falconry, IQ/A, default(IQ - 5, "SK:Animal Handling (Raptors)" - 3), page(B194), cat(_General, Animal)
Mimicry (Animal Sounds), IQ/H, default(IQ - 6, SK:Naturalist - 6, "SK:Mimicry (Bird Calls)" - 6), page(B210),
cat(_General, Animal, Arts/Entertainment, Outdoor/Exploration)
```

You may break the Skills section into different categories by enclosing the category name in angle brackets on its own line. In the sample above, a category called Animal is defined. All of the trait definitions that follow the category (until a new one is encountered) will include that category in their CAT() tags, in addition to any other categories that may already be in the CAT() tag. This allows users to look for traits by category, and allows you to define any categories you need, while not having to include the trait definition under each different category that applies in the data file.

You should not include a colon (:) in the category name, because that has a special meaning. You should also not include any additional angle brackets (< or >) or commas (,).

Spells

The [SPELLS] section allows you to define new spells, or to redefine spells that have been defined in previously loaded files. (Note: It's considered better practice to use commands (see elsewhere) to replace certain tag values in existing traits, rather than redefining them. Replacing tag values allows for replacing just the specific feature that you need changed, rather than having to include the full trait definition. This means you're less likely to end up with an outdated definition if the original source data is changed.)

To define a new spell, you must specify the spell name followed by any other tags needed to complete the trait definition. The only required part of the definition is the name of the spell.

Following the name should be a comma separated list of any other tags necessary to complete the trait.

To redefine an existing spell, the only requirement is that the spell name you specify be exactly the same as the name of the existing spell you wish to overwrite.

Spells are assumed to be based on IQ and of skill type Hard (IQ/H) by default. If the spell is Very Hard (IQ/VH), or any other type, you should be sure to include a TYPE() tag that specifies this, such as type(IQ/VH). The skill type notation should be the type of the skill as specified in the [SKILLTYPES] section in the Basic Set data file, or elsewhere in your data file. The Basic Set data file defines all the standard types for normal ***GURPS*** skills, but other data files may introduce other types.

Here is a small sample Spells section.

```
[Spells]
<Air:Ai>
Purify Air, type(IQ/H), page(M23, B243), mods(Spells), cat(Air), shortcat(Ai), prereqcount(0), magery(0),
class(Area), time(1 sec.), duration(Instant), castingcost(1), description(Prereq Count: 0)
Create Air, type(IQ/H), needs((Purify Air | Seek Air)), page(M23, B243), mods(Spells), cat(Air), shortcat(Ai),
prereqcount(1), magery(0), class(Area), time(1 sec.), duration(5 sec.#), castingcost(1), description(Prereq
Count: 1 Prerequisites: Purify Air or Seek Air)
Shape Air, type(IQ/H), needs(Create Air), page(M24, B243), mods(Spells), cat(Air), shortcat(Ai),
prereqcount(2), magery(0), class(Regular), time(1 sec.), duration(1 min.), castingcost(1 to 10#),
description(Prereq Count: 2 Prerequisites: Create Air)
```

You may break the Spells section into different categories by enclosing the category name in angle brackets on its own line. In the sample above, a category called Air is defined. All of the trait definitions that follow the category (until a new one is encountered) will include that category in their CAT() tags, in addition to any other categories that may already be in the CAT() tag. This allows users to look for traits by category, and allows you to define any categories you need, while not having to include the trait definition under each different category that applies in the data file.

You should not include any additional angle brackets (< or >) or commas (,) in the category name.

Note that the category shown in the sample above includes a colon, even though you've been told not to use it in category names for any other section. In Spells, Categories are also called Colleges, and the College can have a special, shorter code, to make it easier to reference in certain parts of GCA. You specify this special college code by including it in the category name, after a colon. So, in the sample above, the college code for the Air college is Ai.

Templates

The [TEMPLATES] section allows you to define new templates and meta-traits, or to redefine templates and meta-traits that have been defined in previously loaded files. (Note: It's considered better practice to use commands (see elsewhere) to replace certain tag values in existing traits, rather than redefining them. Replacing tag values allows for replacing just the specific feature that you need changed, rather than having to include the full trait definition. This means you’re less likely to end up with an outdated definition if the original source data is changed.)

To define a new template, you must specify the template name followed by any other tags needed to complete the trait definition. The only required part of the definition is the name of the template. Following the name should be a comma separated list of any other tags necessary to complete the trait.

To redefine an existing template, the only requirement is that the template name you specify be exactly the same as the name of the existing template you wish to overwrite.

Here is a small sample Templates section. Notice that line continuation characters are being used to make the templates easier to read.

```
[Templates]
<Character Templates>
Warrior (Basic Set), displaycost(101), cat(Character Templates - Basic Set),
description(You are a fantasy warrior - a barbarian, knight, swashbuckler;_
        or someone else who lives by  the sword.),
page(B448),
sets(_
        ST:ST = 12,
        ST:DX = 12,
        ST:HT = 12_
        ),
adds(_
        SK:Armoury (Melee Weapons) = 1pts,
        SK:Shield (Shield) = 4pts _
        ),
select1(text("If you have not already chosen one, please choose a native Language."), pointswanted(atleast -
3, upto 0), itemswanted(upto 2),
        list(_
                #newitem(_
                        AD:English, 2/4, displaycost(0), page(B24), upto(3 LimitingTotal),
                mods(Language), levelnames([None], Broken, Accented, Native),
        cat(Language, Language Spoken, Language Written, Social Background),
        initmods(Native Language, -6, gives(=+1 to ST:Native Languages),
                        formula(-@if(AD:Language Talent > 0 then 4 else 6)), forceformula(yes),
                        group(Language), page(B23)), taboo(Native Languages > 1)_
                        ) #codes(upto 3, downto 3),
                AD:Language - Native #codes(upto 3, downto 3),
                AD:Language - Native (Spoken) #codes(upto 3, downto 3),
                AD:Language - Native (Written) #codes(upto 3, downto 3)_
                )_
        ),
select2(text("If you have not already chosen one, please choose a native Cultural Familiarity."),
pointswanted(0), itemswanted(upto 1),
        list(_
                AD:Cultural Familiarity (Native)_
                )_
        ),
select3(_
        text("Select two weapon skills from the list below. Each will be taken at 8 pts. _
                Typical selections for a Knight would be Broadsword and Lance."),
        pointswanted(16), itemswanted(2),
        list(_
                SK:Axe/Mace #codes(upto 8pts, downto 8pts),
                SK:Broadsword #codes(upto 8pts, downto 8pts),
                SK:Jitte/Sai #codes(upto 8pts, downto 8pts),
                SK:Lance #codes(upto 8pts, downto 8pts),
                SK:Main-Gauche #codes(upto 8pts, downto 8pts),
                SK:Polearm #codes(upto 8pts, downto 8pts),
                SK:Rapier #codes(upto 8pts, downto 8pts),
                SK:Saber #codes(upto 8pts, downto 8pts),
                SK:Shortsword #codes(upto 8pts, downto 8pts),
                SK:Smallsword #codes(upto 8pts, downto 8pts),
                SK:Staff #codes(upto 8pts, downto 8pts),
                SK:Two-Handed Axe/Mace #codes(upto 8pts, downto 8pts),
                SK:Two-Handed Sword #codes(upto 8pts, downto 8pts),
                SK:Whip #codes(upto 8pts, downto 8pts)_
                )_
        )

<Racial Templates>
Dwarf (Basic Set), displaycost(35), cost(15),
        cat(Racial Templates - Basic Set),
        description(Dwarves might be only 2/3 as tall as humans, but they are much longer-lived, _
                with a nose for gold and a flair for all forms of craftsmanship. _
                Dwarves often live in underground halls, and their eyes are adapted to dim light. _
                Many dwarves have Greed or Miserliness, but these are *not* racial traits.),
        page(B261),
        race(Dwarf),
        noresync(yes),
        owns(yes),
        locks(yes),
        hides(yes),
        gives(_
                +1 to ST:HT,
                -1 to ST:Size Modifier,
                +1 to ST:Will_
                ),
        adds(_
                AD:Artificer=1,
                AD:Detect (Gold)=1 with "Vague, -50%, group(Detect)",
                AD:Extended Lifespan=1,
                AD:Night Vision=5_
```

```
              )
```

Yes, that is a small sample, including one character template and one racial template. Templates are the most complex traits you can make in GCA, and in data files.

You may break the Templates section into different categories by enclosing the category name in angle brackets on its own line. In the sample above, two categories are created, called Character Templates and Racial Templates. All of the trait definitions that follow the category (until a new one is encountered) will include that category in their CAT() tags, in addition to any other categories that may already be in the CAT() tag. This allows users to look for traits by category, and allows you to define any categories you need, while not having to include the trait definition under each different category that applies in the data file.

You should not include a colon (:) in the category name, because that has a special meaning. You should also not include any additional angle brackets (< or >) or commas (,).

Template Types

There are two types of templates in ***GURPS***, and therefore in GCA: character and racial.

Character templates are like help wizards that guide you through the selection of various traits for your character. These templates will not themselves be composed of other traits. In other words, a character template is a simple, directed way for the user to add traits and adjust the values of the character, but the end result is basically no different than if the user had added those traits without the template. If you remove the template later, the traits added from using it remain on the character.

Racial templates, also used for meta-traits, are a single trait (such as Dwarf) that is made up of multiple other component traits (such as Extended Lifespan and Night Vision). These templates include those other traits, and if you remove the template, the component traits are removed as well.

Note that the features that make a template function like a template are based on the tags that are used, such as ADDS() and CREATES(). However, these tags may be used on almost any trait type. Including a template in the [TEMPLATES] section of the data file is largely organizational, not functional; traits that make use of template features can be included as Advantages, Disadvantages, or even Equipment.

Equipment

The [EQUIPMENT] section allows you to define new equipment items, or to redefine equipment items that have been defined in previously loaded files. (Note: It's considered better practice to use commands (see elsewhere) to replace certain tag values in existing traits, rather than redefining them. Replacing tag values allows for replacing just the specific feature that you need changed, rather than having to include the full trait definition. This means you're less likely to end up with an outdated definition if the original source data is changed.)

To define a new equipment item, you must specify the equipment item name followed by any other tags needed to complete the item definition. The only required part of the definition is the name of the equipment item. Following the name should be a comma separated list of any other tags necessary to complete the item.

To redefine an existing equipment item, the only requirement is that the equipment item name you specify be exactly the same as the name of the existing equipment item you wish to overwrite.

Here is a small sample Equipment section.

```
[Equipment]
<Basic Set - Melee Weapons>
Axe, techlvl(0), break(0), lc(4), basecost(50), baseweight(4), page(B271),
    mods(Equipment, Melee Quality, Cutting Class Quality), damage(sw+2), damtype(cut),
    reach(1), parry(0U), minst(11), skillused(Axe/Mace, DX-5, Flail-4, Two-Handed Axe/Mace-3),
    cat(Basic Set, Basic Set - Melee Weapons, _Melee Weapons),
    description(TL:0 LC:4, Dam:sw+2 cut Reach:1 Parry:0U ST:11 Skill:Axe/Mace)
Pick, techlvl(3), break(0), lc(4), basecost(70), baseweight(3), page(B271),
    mods(Equipment, Melee Quality, Crushing/Imp Class Quality), damage(sw+1), damtype(imp),
    reach(1), parry(0U), minst(10), notes([2]), skillused(Axe/Mace, DX-5, Flail-4,
    Two-Handed Axe/Mace-3), cat(Basic Set, Basic Set - Melee Weapons, _Melee Weapons),
    itemnotes({May get stuck; see Picks (p. B405).}),
    description(TL:3 LC:4, Dam:sw+1 imp Reach:1 Parry:0U ST:10 _
    Skill:Axe/Mace Notes: [2] May get stuck; see Picks (p. B405).)
Thrusting Broadsword, techlvl(2), break(0), lc(4), basecost(600), baseweight(3), page(B271),
    mods(Equipment, Melee Quality, Sword Class Quality),
    cat(Basic Set, Basic Set - Melee Weapons, _Melee Weapons),
    newmode(_
            Swing, damage(sw+1), damtype(cut), reach(1), parry(0), minst(10),
            skillused(Sword!, Broadsword, DX-5, Force Sword-4, Rapier-4, Saber-4,
            Shortsword-2, Two-Handed Sword-4)_
            ),
    newmode(_
            Thrust, damage(thr+2), damtype(imp), reach(1), parry(0), minst(10),
            skillused(Sword!, Broadsword, DX-5, Force Sword-4, Rapier-4, Saber-4,
            Shortsword-2, Two-Handed Sword-4)_
            ),
    description(TL:2 LC:4, [Mode:swing Dam:sw+1 cut Reach:1 Parry:0 ST:10 Skill:Broadsword],
    [Mode:thrust Dam:thr+2 imp Reach:1 Parry:0 ST:10 Skill:Broadsword])
```

You may break the Equipment section into different categories by enclosing the category name in angle brackets on its own line. In the sample above, a category called Basic Set - Melee Weapons is defined. All of the trait definitions that follow the category (until a new one is encountered) will include that category in their CAT() tags, in addition to any other categories that may already be in the CAT() tag. This allows users to look for traits by category, and allows you to define any categories you need, while not having to include the trait definition under each different category that applies in the data file.

You should not include a colon (:) in the category name, because that has a special meaning. You should also not include any additional angle brackets (< or >) or commas (,).

Modifiers

The [MODIFIERS] section allows you to specify enhancements and limitations for traits, as well as other modifiers that may be applicable to traits or modifiers.

To define a new modifier, you must specify the name followed by any other tags needed to complete the definition. The only required parts of the definition are the name and the cost of the modifier. Following the name and cost should be a comma separated list of any other tags necessary to complete the definition.

Here is a small sample Modifiers section.

```
[Modifiers]
<_Attack Limitations>
Armor Divisor, -30%/-50%/-70%, levelnames(0.5, 0.2, 0.1), upto(3), page(B110),
        gives(=+@indexedvalue(me::level, 0.5, 0.2, 0.1) to owner::armordivisor)
Bombardment, -5%/-10%, levelnames(Skill 14, Skill 12, Skill 10, Skill 8), upto(4), page(B111)
Dissipation, -50%, page(B112)

<Bestial>
Includes Odious Personal Habit, -5, group(Bestial), page(B124)

<Chronic Pain>
Interval: 1 hour, *0.5, shortname(1 hour), page(B126)
Interval: 2 hours, *1, shortname(2 hours), page(B126)

<Armor>
Armor Quality: Cheap, -0.6 CF, gives(-1 to owner::dr), page(LT109), shortname(Cheap)
```

Modifiers are broken into modifier groups in the same way that traits are broken into categories, by including the group name in angle brackets before the definitions of the modifiers which belong in that group.

Modifier groups are essential to how GCA displays modifiers to users when they're looking to apply modifiers to their traits. GCA generally only displays modifiers that are listed in a trait's MODS() tag, as well as any modifier groups that begin with an underscore (which is used as shorthand for a generally applicable group of modifiers). GCA also displays the EQUIPMENT group for equipment items.

As with traits, a modifier name may include a name extension portion.

The cost section of the modifier definition may take several different forms but is generally written as found in ***GURPS*** material. You may specify percentage costs, flat costs, CF costs, or multipliers. If the cost is leveled, include as many costs as necessary, separated by slashes. GCA will use the difference between the last costs given as the cost for any levels beyond those specified.

Groups

The [GROUPS] section allows you to specify arbitrary groups of traits, for use with pre-req checking or for building selection lists for various options. All items in each group must be trait names and should include appropriate prefix tags. Each new group is specified with angle brackets, like specifying a category in a trait section. Each line after the group is specified should be a new trait in the group.

Here is a small sample Groups section.

```
[GROUPS]
<Conducting>
SK:Musical Instrument
SK:Singing

<Appeal>
ST:Appealing
ST:Unappealing
```

As you can see, two groups will be created, and each of the groups has two traits specified: two skills in the first, two attributes in the second. You are not required to have all traits be of the same type, although that is frequently the case. You may specify as many traits as is needed for the group, one trait per line.

You may specify name extensions as well, if desired, even though none are shown in our example.

You will often want to use the #GROUPLIST directive to create comma-separated lists for use with #CHOICELIST or SELECTX() from Groups.

Lists

The [LISTS] section allows you to specify arbitrary lists of items, for use in a variety of places in GCA. Lists are very similar to Groups, except that each list item may be any desired text. Lists do not need to contain trait names, because they are not expected to be used in the same way.

You will often want to use the #LIST directive to create comma-separated lists for use with #CHOICELIST or SELECTX() from Lists. When doing so, remember that your list items may contain commas, so be sure to use the appropriate flag to have the output appropriately contained for the intended use.

Bonus Classes

BonusClasses exist to provide rules for limiting bonuses from defined "classes" of bonuses.

```
[BonusClasses]
Talents, stacks yes upto 4 best 1
```

Bonuses include themselves in a BonusClass, and the rules for the classes are defined in the data files. Support is currently specific and limited, but allows for limiting the total bonus levels applied from a class of bonuses to an item, or might prevent stacking, and so forth.

Bonus Classes are applied only to addition/subtraction bonuses (whether variable or static) affecting only points or levels.

Here's how you define entries for the [BONUSCLASSES] block in the data files:

> CLASSNAME, [{ AFFECTS | APPLIESTO } { LEVELS | POINTS }] [STACKS { YES | NO }][UPTO X][DOWNTO Y][BEST A][WORST B]

X, Y, A, and B can be expressions, and they'll be sent to the Solver, but you can't use ME:: references because Bonus Classes aren't tagged items.

AFFECTS and APPLIESTO (one word!) are synonyms, use the one you're comfortable with, but don't use both. AFFECTS LEVELS means the class applies to bonus levels only. AFFECTS POINTS means the class applies to bonus points only. The default is AFFECTS LEVELS, so that's what will happen if the AFFECTS clause isn't specified.

STACKS specifies if multiple bonuses from the same class can apply. STACKS YES is the default behavior in GCA for all bonuses. STACKS NO means only 1 bonus from the class will be allowed (usually the best one, unless you also specify WORST 1).

UPTO specifies an upper limit on the total bonus value that may be applied by all bonuses in the class. If exceeded, GCA will create an Adjustment to apply, which will correct for the excess value. If AFFECTS POINTS, do **not** include "pts" in the value.

DOWNTO specifies a lower limit on the total bonus value that may be applied by all bonuses in the class. If exceeded, GCA will create an Adjustment to apply, which will correct for the excess value. If AFFECTS POINTS, do **not** include "pts" in the value.

BEST specifies the maximum number of bonuses that may be applied, and that the highest values up to that number should be used.

WORST specifies the maximum number of bonuses that may be applied, and that the lowest values up to that number should be used.

For example:

```
[BonusClasses]
Talents, stacks yes upto 4 best 2
Acrobatics, stacks no worst 1
```

The Talents bonus class allows stacking as usual, but limits the total bonus from Talents to 4, and only allows the best 2 bonuses (which, if totaling over 4, will have an adjustment created so that the net bonus is 4).

The Acrobatics class doesn't allow bonuses to stack, and only wants the worst possible non-zero bonus to apply. Rough.

Notes: This can get conceptually confusing. Remember that any single bonus will only be applied **once**, no matter the number of classes to which it belongs. If a bonus is excluded for any reason by **any** BonusClass rule, it will not be applied **at all,** regardless of how many other classes it might belong to that didn't exclude it.

See also Bonus Classes in GIVES().

Wizards

These are template-like traits, and the trait data is structured that way, but categories are required to be in a specific format.

Each <CATEGORY> is the name of a trait type, as per the sections of the data file: ATTRIBUTES for attribute wizards, ADVANTAGES for advantage wizards, etc. Currently, support exists in GCA only for the <ATTRIBUTES> category, and currently only for a single wizard there.

```
[WIZARDS]
<Attributes>
New Attribute, noresync(yes%Typealiaslist%),_
x(_
        #InputToTagReplace(Please enter the name of this Attribute:, name, ,New Attribute),_
        #ChoiceList(_
                name(Type),_
                title(Type of Attribute?),_
                text(What type of Attribute do you want?),_
                picksallowed(1),_
                method(bynumber),_
                list(_
                        {A primary attribute (shown in the same listings as DX and IQ)},_
                        {A secondary attribute (not generally displayed, but not hidden)},_
                        {A helper attribute (hidden, used primarily for calculations)}_
                ),_
                aliaslist(_
                        {),step(1), maxscore(1000000), minscore(0), up(5), down(-5), mainwin(20), disadat(-
1), display(yes},_
                        {),step(1), maxscore(1000000), minscore(0), up(5), down(-5), disadat(-1),
display(yes},_
                        {),hide(yes), display(no},_
                )_
        )_
        #edit_
)
```

Wizards are not maintained in the library's trait lists. They are structured like a trait but are not handled like one within the library. You cannot refer to wizards as a trait type and expect valid results. When GCA uses the wizard, it is injected into the character as if it was a trait, and at that time all the trait handling features would be handled normally by the character.

TAG DETAIL INFORMATION

This section will cover the information that you need to write the tags used for the various trait types and modifiers in GDFs. GCA handles many more tags than are covered here; these tags are only those that are used to define traits and modifiers in data files.

Where a specific format is necessary for the tag, we will show a template for the tag like this:

> ADDMODS(MODGROUP:MODNAME TO TARGETTRAIT [, MODGROUP:MODNAME TO TARGETTRAIT])

This uses the usual formatting you should be used to by now, for keywords and variables. Optional portions are enclosed in square brackets, but the square brackets are there only to indicate that optional parts are available, they are not part of the tag.

Math enabled tags are specified.

Flag tags are specified. A flag tag is one that is considered active regardless of the actual value specified by the tag. The only way to ensure no active value is an empty tag, or to not include the tag. In most cases, the convention is that YES is used as the value, such as locks(yes), and the tag is not included when not applicable. Sometimes flag tags are converted to more full-featured service, and in this case abiding by convention can prevent odd behavior as additional values suddenly become valid for the tag, providing for different functions.

Damage modes

A tag that is mode-specific contains within it the data for one or more damage modes, which may have different values for each mode. The mode data is separated by pipe (|) characters. Care should be taken when writing mode-specific tags not to get mode data in the wrong order. Alternatively, mode data may be specified one mode at a time by using NEWMODE() tags.

If GCA calculates values for mode-specific tags, creating CHARX versions of the tag, then in most cases those tags also support the use of special case substitutions, which function like special variables available for use within the mode-specific tag:

%CURMODE	replaced by the current attack mode number.
$MODETAG(TAG)	allows access to the mode-specific tag value for the current mode of the TAG specified.

Mode-specific tags are specified.

Custom Tags

GCA supports the ability for user-created trait tags to be calculated or solved before their value is returned to a request. By including a $ or # at the end of the tag's name, GCA will run the value enclosed in that tag through the text-only function solver (for $), or the normal full numeric solver (for #), before returning the result.

For example, if you were to include a tag called HALFVAL#() on traits, and included an expression, such as halfval#(me::level/2), and then referenced it elsewhere, either on another trait or on a character sheet, GCA would return the solved value when it looked up the value for halfval#; in the example, if level was 10, it would return 5, instead of the expression text.

Note: The tag name must include the $ or #; it is officially part of the tag name. You cannot use a tag with one name, then try referencing it with $ or # to get a certain type of processing, as that will not work. It's all or nothing here.

Pre-Defined Tags

Tags that GCA is already using for various trait features and behaviors. These are generally where you'll include most data for traits.

Acc()

Applies to: traits with damage modes

The Acc value for the weapon or attack. This should be a simple numeric value, sometimes with simple suffix text. If the data is of the format X+Y, X is considered the value, and +Y is considered the suffix.

This tag is mode-specific.

AddMode()

Applies to: modifiers that add new damage modes

Allows a modifier to add a mode to the owning trait. (A modifier to a modifier with ADDMODE() would apply it to the root owning trait, not to the owning modifier.)

```
ADDMODE(MODE NAME[, TAGLIST ])
```

If specifying more than one mode, they should be separated with | characters, and for safety should be enclosed in braces, like so:

```
addmode( { MODE1NAME [, TAGLIST1 ] } | { MODE2NAME [, TAGLIST2 ] } )
```

Mode names must be unique. If the mode name specified already exists on the item, the data provided will replace the data for that mode, instead of another mode of the same name being created.

All data required for a mode must be specified, as data from previous modes will not be carried through. There are two special case variables that can be used as data for a new mode tag, to get data from previous modes:

%COPYPREV	will copy the value for this mode from the immediately preceding mode.
%COPYFIRST	will copy the value for this mode from the first mode.

Each value will be replaced in place with its value, so it may be used as part of a longer expression.

Be aware that GCA has no way of knowing what the original data was once it's been adjusted with this tag. You can't just delete the modifier, or change some of the ADDMODE() data, and expect GCA to restore the original modes for you.

When a modifier uses ADDMODE(), GCA will now calculate a number of tag mode expressions for the mode being added (as best it can, in a similar fashion to how they're calculated if a bonus is applied). The tags listed here will be evaluated: ACC(), ARMORDIVISOR(), DAMAGEBASEDON(), DAMTYPE(), LC(), MINST(), MINSTBASEDON(), PARRY(), RADIUS(), RANGEHALFDAM(), RANGEMAX(), RCL(), REACH(), ROF(), SHOTS(), and SKILLUSED(). Bear in mind that they will not necessarily be evaluated as you might expect, for example, having a new mode Reach of %copyfirst+5 with a mode 1 Reach of 1, will not result in a Reach of 6, because GCA does not determine CHARREACH() based on simple math evaluation—you'll see a Reach of 1+5 instead. If you want to evaluate the two into a single number, you'd have to use $eval(%copyfirst+5) to get the Reach of 6. As stated above, for each tag this will be no different than how GCA determines CHARX values, but it gets put into the base tag, not the CHARX version, so exactly what happens will vary by the tag being evaluated.

Processing of ADDMODE() tags DAMAGEBASEDON(), DAMTYPE(), LC(), MINSTBASEDON(), PARRY(), RCL(), ROF(), and SKILLUSED() will be processed by the text function solver, not the normal solver, which will preserve the text nature of most of these tags, without attempting to solve for a numeric value.

When evaluating ARMORDIVISOR(), ADDMODE() will set the mode tag value to empty when the evaluated value is 0 or 1.

AddMods()

Applies to: traits

This tag allows you to specify modifiers that should be applied to other traits.

> ADDMODS(MODGROUP:MODNAME TO TARGETTRAIT [, MODGROUP:MODNAME TO TARGETTRAIT])

or

> ADDMODS(#NEWMOD(MOD DEFINITION) TO TARGETTRAIT [, MODGROUP:MODNAME TO TARGETTRAIT])

MODGROUP:MODNAME	MODGROUP is the group in which the modifier can be found, and MODNAME is the name and extension (if any) of the modifer to be found. MODGROUP and MODNAME must be separated by a colon, and the whole MODGROUP:MODNAME block can be enclosed in quotes or braces if necessary (such as when it includes the TO keyword or a comma).
TARGETTRAIT	is the name of the trait to which you want to add the modifiers, including prefix tag, and name extension (if any). Enclose it in quotes or braces if necessary.
#NEWMOD(MOD DEFINITION)	allows you to specify a full modifier definition, as is required in some other parts of data files, instead of using a reference to an existing modifier.

You can mix and match MODGROUP:MODNAME or #NEWMOD() blocks. You can also specify multiple modifiers or multiple targets by separating them with commas, but if you do so, you need to enclose the whole block in braces or parens, to be sure that they aren't parsed out separately before the correct time.

Here's a nonsense example:

```
addmods( (Foo:Bar, Flub:Zub) to (SK:Some Skill, SK:Another Skill),
        #newmod(Lub, +1, group(Buz)) to SK:Some Skill )
```

As you can see, you can add multiple modifiers to multiple skills, in multiple blocks, if desired.

Adds()

Applies to: traits

This tag allows you to specify traits that should be added to the character when the current trait is added.

```
ADDS( TRAIT [= VALUE][#DONOTOWN][#NONEEDS][#FORCENEEDS] [#ASCHILD] [ WITH "MODIFIER DEFINITION"[
AND "MODIFIER DEFINITION 2"]][ RESPOND "RESPONSE"])
```

TRAIT	is the trait to be added, using the name and appropriate prefix tag. If the TRAIT name includes any of the keywords or operators, you should enclose it in quotes or braces.
VALUE	is the value desired, as the level. If a point value is desired (for skills and spells only), the value should include the PTS keyword.

By default, GCA considers the = operator to mean “equal to or greater than”, so if you want an exact value, use == for the assignment operator instead.

If you want the added trait to have a specific modifier applied to it, you need to specify the entire definition for the modifier in the WITH block, containing the entire definition within quotes. If you want to assign more than one modifier to the trait, use the AND keyword followed by the next definition, being sure to have a space on each side of the AND. You may include as many AND blocks as you require.

If the trait being added is one that normally asks the user for input, you may automatically assign the value instead of letting the user assign it. You do this with the RESPOND block, which lets you include the intended response.

When adding items from a trait that makes use of #CHOICELIST, you can use RESPOND to select an option from that #CHOICELIST for the user, much as you can make choices for #INPUT. The value of the response for a #CHOICELIST should be an integer representing the option index to select. The index number is based on the order of the options presented *in the data file* for the #CHOICELIST (just as the default options are selected in #CHOICELIST)—do not set the choice desired based on the order presented in the pick dialog, as those options are sorted alphabetically, and may not represent the order that GCA will use to make the selection. If you are setting the response for a #CHOICELIST that expects more than one pick, you may have your response text be a list of integers separated by commas, but remember to enclose the entire list in quotes, not each separate integer (otherwise it will appear to be responses for separate dialogs).

The RESPOND block may use a $ function to process the response text before it gets pushed onto the response stack. You can use this to completely change responses, or to respond with an evaluated expression. If you want to respond with text that includes an unprocessed $ function, escape it as $/FUNCTION() rather than simply $FUNCTION() and GCA will change the $/ to $ before it pushes the responses onto the stack. The $ function processing happens before any quotes or braces are stripped from around the response expression (which itself happens before the response is split into list items if it is a list).

Note that the quotes around the values in the WITH, AND, and RESPOND blocks are required, although curly braces may be used if you prefer them, or if the content of those blocks might include quotes.

The order of the blocks is also important; you must include any desired blocks in the order shown, although you may leave out any blocks you don't need.

If the adding trait has an OWNS(YES) tag, GCA will consider any traits added by an ADDS() tag to be owned by the adding trait. This means that the trait with the ADDS() tag is effectively operating as a template. GCA will also add the added traits to the NEEDS() tag of the adding trait.

The special #DONOTOWN directive tells GCA that you don't want the added trait to be owned by the adding trait. You must include this for each trait you don't want automatically owned. If included, the trait will also not be hidden or locked if the adding trait also has HIDES(YES) or LOCKS(YES) applied.

The special #NONEEDS directive tells GCA that you don't want the added traits to be added to the NEEDS() tag of the adding trait. If #DONOTOWN is applied, #NONEEDS is not required.

The special #FORCENEEDS directive tells GCA that it should allow auto-adding of NEEDS() for this trait when it's added to the character. The normal behavior for templates adding traits in an ADDS() or ADDSORINCREASES() is to disable auto-adding of needs items since that may conflict with other elements of the template that have not yet been processed, and could result in unintended duplicate traits.

The special #ASCHILD directive tells GCA that the added trait should be added as a child. Using #ASCHILD automatically forces #DONOTOWN as well, so be aware that the child will not be owned and will not appear in the NEEDS() automatically. As with #NONEEDS and #DONOTOWN you should include #ASCHILD in the main clause of the tag data, before any RESPOND or WITH block, else it be considered part of the other clauses and not work correctly; sticking it into the name section is usually good.

You may add as many traits as desired by separating each section with commas.

Example 1, a relatively simple ADDS() tag:

```
adds(_
	SK:Armoury (Melee Weapons) = 1pts,
	SK:Shield (Shield) = 4pts _
	),
```

Example 2, a more complex ADDS() tag, using WITH, AND, and RESPOND blocks:

```
adds(_
	AD:Burning Attack _
			with "Always On, -40%, group(_General)" _
			and  "Aura, +80%, group(_Attack Enhancements)" _
			and  "Melee Attack: Reach C, -30%, group(_Attack Limitations), page(B112),
					gives(=nobase to owner::rangehalfdam$, =nobase to owner::rangemax$,
                              =nobase to owner::reach$, ="C" to owner::reach$)" _
     respond 1,
     AD:Doesn't Breathe _
               with "Oxygen Combustion, -50%, group(Doesn't Breathe)",
     AD:Damage Resistance=10 _
               with "Limited: Heat/Fire, -40%, group(Limited Defense)",
     AD:Injury Tolerance _
               with "Diffuse, +100, group(Injury Tolerance)",
     DI:No Manipulators,
     DI:Weakness (Water)=3 _
               with "Rarity: Common, *2, shortname(Common), group(Weakness)" _
     ),
```

Expanded Features: #Loadout

The special #LOADOUT() directive allows you to specify loadouts that equipment items should be added to automatically. This works like the LOADOUT() trigger tag.

Note that owned items will not be added to loadouts through this command, because owned items are always included with their parent items in loadouts.

If you use the #LOADOUT() special directive to add an item to a loadout, then GCA will automatically apply protection from any armor or shield items so added. This seemed to be the most likely desired action, so it was made the default. If you do NOT want that to happen, include #NOARMOR as part of the loadout name inside the #LOADOUT() directive for each loadout to not receive protection.

If you are adding a shield, you can use #ARC() as part of the loadout name to specify the protected arc. At this time, the supported arcs are none (the default), back, left arm, and right arm.

Example:

```
Loadout Tester (Adds),
  displaycost(0),
  cost(0),
  page(User),
  noresync(yes),
  adds(_
               {EQ:Backpack, Small #loadout(Pack)}=1,
               {EQ:Blanket #loadout(Pack)}=2,
               {EQ:Canteen #loadout(Pack)}=1,
               {EQ:Leather Jacket #loadout(Pack, Combat)}=1,
               {EQ:Heavy Leather Leggings #loadout(Pack, Combat)}=1,
               {EQ:Gauntlets #loadout(Pack #noarmor, Combat)}=1,
               {EQ:Small Shield #loadout(Pack #arc(back), Combat #arc(left arm) ) }=1,
               {EQ:Steel Breastplate #loadout(pack #noarmor)}=1_
     )
```

This template will apply all items to the pack loadout, and some of the armor to the combat loadout. The Steel Breastplate is not being worn, so it's flagged as #NOARMOR in the pack loadout where it resides. The Small Shield is worn on the back in the pack loadout but on the left arm in the combat loadout. And since gauntlets are annoying to wear casually, they're also not being worn in the pack loadout.

You can also use this feature to assign protection for the virtual 'All unassigned items' loadout by leaving the loadout name portion blank, or by explicitly referencing a blank loadout using empty quotes "".

AddsOrIncreases()

Applies to: templates

This tag works somewhat like ADDS() or SETS(), but it first looks for the trait on the character, and if found, increases it by the given value. If not found, GCA tries to add the trait to the character at the given value like an ADDS().

The special #ASCHILD and #FORCENEEDS directives are also supported here (see Adds() for details).

Given that known attributes always exist, you'd use this with attributes as 'increase by this amount', so to add 1 to ST you'd use ST:ST=1.

For example:

```
addsorincreases(_
        {ST:Perception}= 1,_
        {AD:High Manual Dexterity}= 1,_
        {SK:Stealth}= 8pts,_
        {SK:Lockpicking}= 2pts,_
        {SK:Forced Entry}= 1pts_
)_
```

In this example, Perception will be increased by 1, because attributes should always exist; High Manual Dexterity will be increased by 1 if the character already has it or it will be added if not. Likewise, each of the skills will be added at the given point values, or if the character already has those skills, will increase the points spent on them by the given amount.

This has several aliases because I couldn't settle on the name: ADDORINCREASE(), ADDSORSETS(), and ADDORSET().

This works inside TRIGGERS() or is processed in the standard tag list after ADDS().

See also the Adds() tag **Expanded Features: #Loadout** section for more information, as #LOADOUT() is also supported here.

Age()

Applies to: templates

This tag allows you to specify an age, which will be inserted into the character's Age field. This will replace any existing age that may already have been added to the character.

Appearance()

Applies to: templates

This tag allows you to specify an appearance, which will be inserted into the character's Appearance field. This will replace any existing appearance that may already have been added to the character.

ArmorDivisor()

Applies to: traits with damage modes

The Armor Divisor value for the weapon or attack. This should always be a simple numeric value.

This tag is mode-specific.

Base()

Applies to: advantages, perks, disadvantages, quirks

If BASE() exists, GCA will calculate it, and add it on to the final level of the advantage as a 'base value' instead of the normal base value of 0. For example,

```
base(10)
```

for an advantage would mean that taking one level of the advantage would give it level 11.

Math enabled.

BaseCost()

Applies to: equipment

The dollar cost of one piece of the equipment. This should always be a simple numeric value.

BaseCostFormula()

Applies to: equipment

This allows you to specify a formula to use to determine the base cost of the item before count or children. If used, this replaces the value of the base cost as set by basecost() with the calculated result.

BaseQty()

Applies to: equipment

The initial count or quantity of the equipment item when first taken by the user. This may be a simple value such as 10, or a math expression.

This tag is math enabled.

BaseValue()

Applies to: attributes

The initial score of the attribute before the user has changed it. This may be a simple value such as 10, or a math expression.

This tag is math enabled.

BaseWeight()

Applies to: equipment

The weight of one piece of the equipment. This should always be a simple numeric value.

BaseWeightFormula()

Applies to: equipment

This allows you to specify a formula to use to determine the base weight of the item before count or children. If used, this replaces the value of the base weight as set by baseweight() with the calculated result.

BlockAt()

Applies to: traits

This tag allows you to specify the normal Block score when using the trait to block. For example

```
blockat(@int(%level/2)+3)
```

uses one-half of the value of the skill's level (dropping fractions), plus 3, for the trait's Block.

This tag is math enabled.

BodyType()

Applies to: templates

This tag allows you to specify a body type, which will be inserted into the character's Body Type field. This will replace any existing body type that may already have been added to the character.

Break()

Applies to: traits with damage modes

The Break value for the weapon or attack. This should always be a simple numeric value.

This tag is mode-specific.

Bulk()

Applies to: traits with damage modes

The Bulk value for the weapon or attack. This should always be a simple numeric value.

This tag is mode-specific.

Cat()

Applies to: traits

This tag allows you to specify the categories to which the trait belongs. Categories are also added to this tag by GCA for the category marker under which it is found in the data file.

Categories are separated by commas, and this tag does not allow for quotes or braces, so commas and other restricted characters are not allowed.

CharHeight()

Applies to: templates

This tag allows you to specify a height, which will be inserted into the character's Height field. This will replace any existing height that may already have been added to the character.

CharWeight()

Applies to: templates

This tag allows you to specify a weight, which will be inserted into the character's Weight field. This will replace any existing weight that may already have been added to the character.

ChildOf()

Applies to: traits

This tag allows for specifying a trait which the added item is to be made a child of. Usually, you'll want to add that item with `ADDS()` or something, first, or otherwise ensure that it's likely to be on the character already. If the trait being targeted by `CHILDOF()` is not found, the item is still added to the character, just not as a child.

ChildProfile()

Applies to: parent traits

This tag allows you to specify special handling by the parent trait, for the costs of child traits, in a parent/child relationship. If this tag is missing or has a value of 0, the normal behavior applies (the full costs of the child traits are included in the total cost of the parent trait.) If this tag has a value of 1, the child traits are treated as Alternative Attacks (p. B61), and their costs are adjusted appropriately before being included in the total cost of the parent trait.

Collapse()

Applies to: parent traits, owner traits

COLLAPSE() allows you to indicate if children or components are intended to be shown or not. This is currently a flag tag. GCA currently allows for changing the collapsed state of a parent using the right-click menu, or the Traits menu, when an applicable trait is selected.

CollapseMe()

Applies to: parent traits, owner traits

Allows you to use a text formula to set the value of the COLLAPSE() tag.

The TextFunctionSolver is used to return a text result. If the result is empty ("", nothing), the COLLAPSE() tag is removed, otherwise the result is placed into the COLLAPSE() tag.

(This uses the text functions in case I decide to add special cases to COLLAPSE() later, changing it from a flag tag to a value dependent one.)

Conditional()

Applies to: traits and modifiers

This tag allows you to specify conditional bonuses granted by the trait. These bonuses will not be included in the value of the target trait, but a message will be available for printing on the character sheet and inside GCA about when the bonus applies.

```
CONDITIONAL([=] BONUS to TARGET[::TAG] [ UPTO LIMIT ][ WHEN "CONDITION" ][ LISTAS "BONUS TEXT"])
```

The optional = marker allows you to specify that the bonus being applied is a single bonus, not to be applied on a per-level basis. Without the =, bonuses are applied per level of the trait by default.

BONUS	is the bonus to be applied, whether it's positive or negative, or some other text in certain cases. BONUS is math enabled.
TARGET	is the trait to which the bonus should be applied. The TARGET name should be enclosed in quotes if it includes any restricted characters. In addition to traits, the TARGET may also be a group, category, class, college, or a variety of other special keywords.
TAG	this may only be included if the TARGET is a trait. Not all tags may receive bonuses.
UPTO LIMIT	this block allows you to specify a maximum value for the bonus. LIMIT is math enabled.
WHEN "CONDITION"	this block allows you to specify a short text description of when the conditional bonus is applicable.

LISTAS "BONUS TEXT" this block allows you to specify the text listed when GCA or a character sheet displays the reason a particular bonus is being applied. There are several special case substitution variables available for use in a LISTAS block: %VALUE% for the current value of the bonus; %STRINGVALUE% for the string value of the bonus; %NAME% for the display name of the trait granting the bonus, and %FULLNAME% to get the full name of the granting trait rather than the display name.

You must be careful to ensure that there is a space to either side of each of the various keywords you use in the tag, or the tag will not be parsed correctly.

You may add as many bonuses as desired by separating each bonus section with commas.

Example:

```
conditional(_
    +2 to SK:Stealth when "perfectly still, unless clothed",
    +1 to SK:Stealth when "moving, unless clothed",
    +1 to SK:Stealth when "perfectly still, and clothed"_
    )
```

Cost()

Applies to: traits other than attributes, skills, and spells; modifiers

This tag specifies the cost, or cost progression, of the trait. Usually, these values are specified in the definition without explicitly using this tag.

Note: In an unfortunate oversight early in the development life of GCA, COST() was also used as the final dollar cost of equipment items. Usually this causes no issues because the types of traits are quite different, but you should be aware of the overlap.

CostFormula()

Applies to: equipment

This tag allows for specifying an expression that should be evaluated to determine the cost of an item. If a FORMULA() tag is included, the cost will always be calculated based on the given formula. This tag overrides much of the normal behavior of costing an item, overriding anything calculated using modifiers and child items. For this reason, it may be better to use BASECOSTFORMULA() for the base cost, unless you actually want a full override for some reason.

Count()

Applies to: equipment

This tag specifies the number of items the equipment item includes.

CountAsNeed()

Applies to: traits

This tag allows for an item to be excluded as a possible means of satisfying a prerequisite item for another trait's needs checking.

If the trait should not be counted as a valid prerequisite item in any case, include COUNTASNEED(NO) in the tag list.

If the trait may be counted in certain cases, additional detail may be included. In this case, the value of the tag may be a list of special identifiers for which it is valid to count the item as a prerequisite, but any item not including a listed identifier may not count the item as a prerequisite.

An item may use the corresponding IDENT() tag to specify one or more identifiers for purposes of needs checking, such as ident(Summoner, Priest), in which case any COUNTASNEED() tag that specifies Summoner or Priest, or both, would be a possible valid prerequisite for the item with that IDENT().

CountCapacity()

Applies to: traits

The number of child items the item can contain.

When GCA processes the item, it will create these tags: COUNTCAPACITYLEVEL(), which shows the percentage of the current capacity used; OVERCOUNTCAPACITY(), which is just YES if the item currently exceeds its capacity; OVERCOUNTCAPACITYBY(), which contains the amount by which capacity is exceeded.

Creates()

Applies to: traits

This tag allows you to create new traits that should be added to the character when the current trait is added.

> CREATES({TRAIT DEFINITION} [= VALUE][#DONOTOWN][#NONEEDS][#ASCHILD] [WITH "MODIFIER DEFINITION" [AND "MODIFIER DEFINITION 2"]])

TRAIT DEFINITION	is the trait to be created on the character, using the name and appropriate prefix tag. The TRAIT DEFINITION should generally be enclosed in braces, but if it's simple enough to not include quotes, it may be enclosed in quotes, instead.
VALUE	is the value desired, as the level. If a point value is desired (for skills and spells only), the value should include the PTS keyword.

By default, GCA considers the = operator to mean "equal to or greater than", so if you want an exact value, use == for the assignment operator instead.

If you want the added trait to have a specific modifier applied to it, you need to specify the entire definition for the modifier in the WITH block, containing the entire definition within quotes. If you want to assign more than one modifier to the trait, use the AND keyword followed by the next definition, being sure to have a space on each side of the AND. You may include as many AND blocks as you require.

Note that the quotes around the values in the WITH and AND blocks are required, although curly braces may be used if you prefer them, or if the content of those blocks might include quotes.

The order of the blocks is also important; you must include any desired blocks in the order shown, although you may leave out any blocks you don't need.

If the adding trait has an OWNS(YES) tag, GCA will consider any traits added by an ADDS() tag to be owned by the adding trait. This means that the trait with the ADDS() tag is effectively operating as a template. GCA will also add the added traits to the NEEDS() tag of the adding trait.

The special #DONOTOWN directive tells GCA that you don't want the added trait to be owned by the adding trait. You must include this for each trait you don't want automatically owned. If included, the trait will also not be hidden or locked if the adding trait also has HIDES(YES) or LOCKS(YES) applied.

The special #NONEEDS directive tells GCA that you don't want the added traits to be added to the NEEDS() tag of the adding trait. If #DONOTOWN is applied, #NONEEDS is not required.

The special #ASCHILD directive tells GCA that the added trait should be added as a child. Using #ASCHILD automatically forces #DONOTOWN as well, so be aware that the child will not be owned and will not appear in the NEEDS() automatically. As with #NONEEDS and #DONOTOWN you should include #ASCHILD in the main clause of the tag data, before any RESPOND or WITH block, else it be considered part of the other clauses and not work correctly; sticking it into the name section is usually good.

You may add as many traits as desired by separating each section with commas.

Expanded Features: #Loadout

The special #LOADOUT() directive allows you to specify loadouts that equipment items should be added to automatically. This works like the LOADOUT() trigger tag. See also the Adds() tag
Expanded Features: #Loadout

section for more information, as #LOADOUT() has additional functionality which is discussed there.

Note that owned items will not be added to loadouts through this command, because owned items are always included with their parent items in loadouts.

Damage()

Applies to: traits with damage modes

This is the damage for the attack. Generally written as a standard GURPS damage code, such as damage(2d-1), damage(thr), or damage(sw+2).

This tag is math enabled.

This tag is mode-specific.

DamageBasedOn()

Applies to: traits with damage modes

If this tag exists, GCA will use the attribute specified within for determining damage for an item or trait. In effect, GCA will replace damage codes of THR or SW with @THR() or @SW(), using the attribute specified within the function.

This tag is mode-specific.

DamageIsText()

Applies to: traits with damage modes

This is a special case damage calculation tag. This is a mode-specific tag, so certain modes may be YES while others are empty (this is a flag-tag, so empty means it doesn't apply, while any other value means it does apply). If DAMAGEISTEXT(YES) is found for a mode, that mode will preserve the text value of the DAMAGE() given, and will calculate only the bonuses that might apply, tacking them on to the end if available.

This tag is mode-specific.

This tag is a flag tag.

DamType()

Applies to: traits with damage modes

This is the type of damage applied by the attack. This is generally a text value specifying the standard damage types, such as damtype(cut) or damtype(imp).

This tag is mode-specific.

DB()

Applies to: traits

This tag specifies the DB value of the trait. This should be a simple numeric value.

DecimalPlaces()

Applies to: equipment

For equipment items to allow for changing the number of decimals kept during calculation of modifiers, allowing items to override the normal 2 digits kept for equipment. Because of other issues in modifier calculation, 4 decimal places is still the max, but you can specify a value from 0 (the default is 2 places) to 4. This may help in certain cases.

For example,

```
decimalplaces(3)
```

would set the places kept to 3.

Default()

Applies to: skills and spells

This tag allows for specifying a list of traits from which the trait may default. As is usual with these lists, items should be separated by commas. You should specify prefix tags where possible. If a trait name includes any math or restricted characters, it should be enclosed in quotes. For example:

```
default(IQ - 6, SK:Naturalist - 6, "SK:Mimicry (Bird Calls)" - 6)
```

This example includes two skill names, the second of which includes parens, which are math characters, so it is enclosed in quotes.

This tag is math enabled, but in a more restricted fashion than normal. GCA expects there to be a clear trait name of some kind at the front of each possible list item, which it will use to show the user what the trait is defaulting from. Math expressions that vary much from that format may not work correctly.

Expanded Features

Parsing should now honor quotes, braces, and parens. Before, it only honored quotes, so if you had a trait name that included a comma within the name extension, it would parse on that unless you put quotes around it. Now, you should only need quotes or braces around traits that have commas that aren't otherwise contained.

Some non-trait defaults are now allowed to pass as valid within GCA, primarily for reasons in the next bit. This shouldn't break anything because such things, on their own, should still evaluate to 0 if they're nonsense, but if they're numeric, they allow for defaults from numbers and such. (Any non-trait item gets a DEFFROMID() of 0, since such things obviously have no IDKey.)

Also added text function solver support, and set it to be processed first thing, so that you can use text functions to completely alter the way that a default may be processed. This means you could do something like this:

```
default( $if(1 = 2 then "SK:Whatever"-2 else "SK:Whatever Else"-6) )
```

to just return the default statement you want, depending on the evaluation of the $If comparison.

@Default()

Support has been added which allows for a complex expression to be used to set the default level. Doing this requires the use of the @DEFAULT() specialty function. Anything inside the @DEFAULT() parens is sent to the full Solver.

The @DEFAULT() function should be at the end of the default item, while at the beginning of it should be the text that will be shown to the user as the deffrom() text. So, it should look something like this:

```
default(Some Expression @default(@max(10, 12))
```

which would give us deffrom(Some Expression) and deflevel(12). That's some pointless math, but that's the idea.

Deflect()

Applies to: traits

Provides the value of any deflect bonus from the trait. Serves as the basis for calculating the CHARDEFLECT() tag, which is used as a bonus to any DB() value when calculating CHARDB().

Description()

Applies to: traits and modifiers

This tag includes a description of the trait.

RTF: GCA supports using RTF text inside this tag, instead of the plain text used in most other places. GCA provides a rudimentary RTF editor to allow you to change text colors and fonts, or to bold or italicize words in the text, but if you want to get fancier you may want to write things in WordPad or another RTF editor and then paste the RTF into this tag.

While GCA internally allows for carriage returns (CR) and line feeds (LF) within this text, the GDF format does **not**. Within the book data, this text must still fall back to a single line after line continuation characters. This is important because RTF frequently includes line feed (LF) characters within the RTF text, usually after paragraph or newline codes.

You can manually replace CR+LF, CR, or LF characters with special codes that will be replaced by GCA with the corresponding characters upon loading the data. Use <CRLF> for CR+LF, <CR> for CR, and <LF> for LF. Be sure that you actually remove the offending control characters after you insert the codes, otherwise your trait will not load correctly.

Display()

Applies to: attributes

This tag allows you to specify whether an attribute is intended to be seen by the user in secondary display areas for attributes. All visible attributes are displayed by default in secondary display areas unless DISPLAY(NO) is included. Since many attributes are used as helper stats or as variables, including DISPLAY(NO) helps limit the number of inapplicable attributes the user must look through.

All primary attributes (ST, DX, etc.) are displayed in the primary display areas regardless of DISPLAY(NO). Inclusion in the primary display areas is generally controlled by the use of the MAINWIN() tag, or in some cases, is simply not able to be disabled.

DisplayCost()

Applies to: system traits

When DISPLAYCOST() exists, the value of the tag will be used instead of the standard displayed cost for the system trait. This tag is removed from the data for the character's version of the trait.

DisplayName()

Applies to: system traits

Similar in intent to the existing DISPLAYCOST() and DISPLAYWEIGHT() tags.

When DISPLAYNAME() exists, the value of the tag will be used instead of the standard return from the DisplayName property for the system trait (usually Full Name/TL). This tag is removed from the data for the character's version of the trait, where DISPLAYNAMEFORMULA() can still be used if desired.

DisplayNameFormula()

Applies to: traits, modifiers

When using the DisplayName property of a Trait (which most trait lists inside GCA use), if DISPLAYNAMEFORMULA() exists, GCA will use that to generate the name being returned, instead of the built-in functions. This is Text Function Solver enabled.

```
displaynameformula( $val(me::name): $val(me::levelname) )
displaynameformula( You resist on a roll of $val(me::levelname) )
```

In conjunction, the BASEDISPLAYNAME() function (for plugin writers), or ME::BASEDISPLAYNAME will return the same info as DisplayName, but will not use the formula, so it can be used within formulas to alter the output that GCA would have generated.

If you use VARS(), you can get significantly more advanced results with reduced confusion. See VARS() below for more.

This has an alias tag of RESKIN(), which should be easier to type. However, because it's an alias, it'll be read correctly, but it will be handled and processed by GCA as DISPLAYNAMEFORMULA(), so it'll only live under the RESKIN() name until the data is loaded, and then it will be gone. However, if you need the tag value for some reason, you can use ::RESKIN to get it.

DisplayScoreFormula()

Allows for customizing the display of score/level values wherever GCA shows them using the standard display method.

Other than affecting what's normally the score or level of the trait, this works just like DISPLAYNAMEFORMULA().

For example, you might click on Advanced next to TL in Campaign Settings, and use the Display Score Builder dialog to change your base TL to 6+2^, and then saved it to a data file; this is what you'd see in that file:

```
Tech Level,basevalue(6),maxscore(12),minscore(0),up(5),down(-5),symbol(TL),round(1),
        mods(Tech Level),mainwin(14),
        displayscoreformula(%front%%value%%back%),scoreback(+2^),
        vars(%front%=$val(me::scorefront), %back%=$val(me::scoreback), %value%=$val(me::score))
```

This uses custom tags* to simplify tracking the bits, VARS() to simplify accessing the parts, and DISPLAYSCOREFORMULA() to combine them into a single whole to display to the user.

* Note that the Display Score Builder supports SCOREBACK() and SCOREFRONT() as custom tags, but SCOREFRONT() has no value in this example, so wasn't saved.

DisplayWeight()

Applies to: system equipment traits

When DISPLAYWEIGHT() exists, the value of the tag will be used instead of the standard displayed item weight. This tag is removed from the data for the character's version of the trait.

Down()

Applies to: attributes

This tag specifies the cost for each step decremented below the base value of the attribute. This may include multiple costs, separated by slashes, if the cost isn't the same for each level. GCA will use the difference between the last two costs as the cost per level for any additional levels beyond the given progression.

All values specified in the DOWN() tag must be simple numeric values.

For example, down(-5) specifies a cost of -5 points per level lowered, while down(-5/-10/-20/-40) specifies costs ranging from -5 points for the first decrement, to -20 points per decrement after the third.

DownFormula()

Applies to: attributes

Allows you to specify a formula to use for calculating the cost of the attribute when lowering the attribute from the base value. Math enabled.

DownTo()

Applies to: traits other than equipment, modifiers

This tag is math enabled.

Languages, Cultures, Advantages, Perks, Features, Disadvantages, Quirks, and Templates; Modifiers

```
DOWNTO(VALUE)
```

This tag specifies the minimum number of levels allowed for the trait.

Skills and Spells

```
DOWNTO(VALUE [PTS])
```

This tag specifies the minimum level allowed for the trait.

If PTS is specified, then the DOWNTO() value is actually the minimum points that may be spent in the skill.

DR()

Applies to: traits

This tag specifies the DR value of the trait. This should be a simple numeric value, possibly with a simple text suffix. In some cases, this may be two such values separated by a slash.

DRNotes()

Applies to: traits

Includes one or more notes about the DR provided by the trait.

```
drnotes( {Split DR: use the lower DR against crushing attacks.} )
```

Each note should be an individual item, and multiple notes should be separated by commas. Enclose notes in braces or quotes if necessary to avoid incorrect parsing.

Features()

Applies to: templates

Acts like a CREATES() to create a new Feature as a component trait of the template.

FencingWeapon()

Applies to: traits

Added support for a FENCINGWEAPON() tag. The value for this tag, should be CHAR::ENCLEVEL such as

```
fencingweapon(char::enclevel)
```

because that will force GCA to create an association between the character and the trait, to ensure that it is recalculated when GCA's encumbrance level changes. The given value of the tag is not actually used.

When this tag exists, and the character's encumbrance level is above 0, GCA will create a CHARFENCINGPENALTY(-X) tag, and add an item to the bonus list for encumbrance level affecting fencing weapon attacks and parries.

GCA will then use the CHARFENCINGPENALTY () value to reduce the CHARSKILLSCORE() for the weapon. It then adds that value *back* before calculating the CHARPARRYSCORE() based on CHARSKILLSCORE(), and then reduces CHARPARRYSCORE() by that amount.

Net result: CHARSKILLSCORE() is reduced by ENCLEVEL for the attack tables, and CHARPARRYSCORE() is reduced by ENCLEVEL for the attack tables, and CHARPARRYSCORE() is not affected by 'double dipping' the encumbrance penalty.

Note that if the PARRY() for a weapon includes the F flag indicating a fencing weapon, GCA will automatically generate this tag for the weapon as part of the calculations.

Flexible()

Applies to: equipment with DR

Armor items denote if they are flexible in the same way the books do: with a * at the end of the DR value. The FLEXIBLE() tag can also be used to denote whether the armor is flexible or not.

> FLEXIBLE(VALUE)

A value of 0, no, or false to mark the armor as **not** flexible. Any other value will cause the armor to be marked **as** flexible but using values of yes or true is recommended for clarity. Using this tag will override any existing * marker.

To apply bonuses to the flexible state, apply a string bonus to FLEXIBLE$ of no or false to turn flexible off, or yes or true to turn flexible on. You can also target any positive adder bonus to turn flexible on, or any negative adder bonus to turn flexible off. The string bonus overrides the adder bonus if both are present.

GCA will calculate a CHARFLEXIBLE() tag based on the * marker or the FLEXIBLE() tag, and any bonuses targeted at FLEXIBLE. If the armor is determined to **not** be flexible CHARFLEXIBLE() will NOT exist and will be deleted if it did exist. Likewise, GCA will add the * to the end of the DR value if flexible and remove it if not flexible.

ForceFormula()

Applies to: modifiers

If FORCEFORMULA(YES) is included in the modifier's tag list, the FORMULA() specified will be used to calculate every level cost for the modifier, not just levels beyond the costs specified in the definition.

This tag is a flag tag.

Formula()

Applies to: traits other than attributes, skills, and spells; modifiers

Traits: This tag allows for specifying an expression that should be evaluated to determine the cost of a trait. If a FORMULA() tag is included, the cost will always be calculated based on the given formula, but a COST() tag still needs to be provided if the trait is to be leveled, because that's what GCA uses to determine that there are multiple levels possible (that cost can be 0/0 or similar). A DISPLAYCOST() tag may also be appropriate.

Equipment: In equipment items, this tag overrides much of the normal behavior of costing an item, overriding anything calculated using modifiers and child items. For these, it may be better to use BASECOSTFORMULA() for the base cost, unless you actually want a full override for some reason. This is also aliased as COSTFORMULA() in equipment items, if you'd like to keep the differentiation clear.

Modifiers: Similar to traits, but the FORMULA() is only used to determine costs beyond the costs specified in the definition of the modifier. If you wish the formula to specify all costs, use FORCEFORMULA(YES) also.

Fortify()

Applies to: traits

Provides the value of any fortify bonus from the trait. Serves as the basis for calculating the CHARFORTIFY() tag, which is used as a bonus to any DR() value when calculating CHARDR().

Gives()

Applies to: traits and modifiers

This tag allows you to specify all bonuses granted by the trait. These bonuses will be included in the value of the target trait.

> GIVES([=] BONUS TO TRAIT[::TAGNAME] [BYMODE [WHERE] TAG COMPARISON VALUE] [UPTO LIMIT][LISTAS "BONUS TEXT"])

The optional = marker allows you to specify that the bonus being applied is a single bonus, not to be applied on a per-level basis. Without the =, bonuses are applied per level of the trait by default.

BONUS	is the bonus to be applied, whether it's positive or negative, or some other text in certain cases. BONUS is math enabled.
TRAIT	is the trait to which the bonus should be applied. The TRAIT name should be enclosed in quotes if it includes any restricted characters. In addition to traits, the TRAIT may also be a group, category, class, college, or a variety of other special keywords.
TAGNAME	this may only be included if the TRAIT is a trait. Not all tags may receive bonuses.
UPTO LIMIT	this block allows you to specify a maximum value for the bonus. LIMIT is math enabled.
BYMODE [WHERE] TAG COMPARISON VALUE	this optional block allows you to specify that the bonus applies on a per-mode basis and may not be universally applicable to the target. See the ByMode section below for details.
LISTAS "BONUS TEXT"	this block allows you to specify the text listed when GCA or a character sheet displays the reason a particular bonus is being applied. There are several special case substitution variables available for use in a LISTAS block: %VALUE% for the current value of the bonus; %STRINGVALUE% for the string value of the bonus; %NAME% for the display name of the trait granting the bonus, and %FULLNAME% to get the full name of the granting trait rather than the display name.

You must be careful to ensure that there is a space to either side of each of the various keywords you use in the tag, or the tag will not be parsed correctly.

You may add as many bonuses as desired by separating each section with commas.

Example:

```
gives(+1 To GR:Perfect Balance, +4 to SK:Immovable Stance)
```

ByMode

This optional block allows you to specify that the bonus applies on a per-mode basis and may not be universally applicable to the target.

```
BYMODE [WHERE] TAG COMPARISON [SUBCRITERIA] VALUE
```

The BYMODE keyword is required but the WHERE keyword is optional (it just helps readability to have it there).

TAG is the mode-enabled tag that you're looking at, such as DAMTYPE.

COMPARISON is the comparison you're making, and uses the same selection as Trait Selectors, which are these:

IS	TAG and VALUE are the same
ISNOT	TAG and VALUE are not the same
INCLUDES	VALUE can be found inside TAG
EXCLUDES	VALUE is not found inside TAG
LISTINCLUDES	TAG is treated as a list, and VALUE is found as one of the list items
LISTEXCLUDES	TAG is treated as a list, and VALUE is not found as one of the list items

SUBCRITERIA is an optional refining of the COMPARISON using keywords to limit to what the COMPARISON is applied. These keywords are ONEOF, ANYOF, ALLOF, or NONEOF. This allows you to specify a list of options to use for the TAG value that satisfy or fail the requirements and allows VALUE to be a list of possible options separated by commas (quotes or braces as needed).

VALUE is the value that you're comparing TAG against.

Right now, this is basically all string/text based, so there's no less than or greater than or such comparisons. I can add something to do that if needed down the line.

If the BYMODE clause exists for the bonus, and the comparison returns TRUE, then the bonus applies for the mode.

Here's an example GIVES() that might be applied by a modifier:

```
=+5 to owner::armordivisor bymode where damtype contains "cut"
```

This bonus to armor divisor is only applied when the owner's mode is something that does cutting damage, so if there's a hammer on the back doing crushing, it will not suddenly get an armor divisor as well.

Here's an example using subcriteria:

```
=+5 to owner::armordivisor bymode where damtype includes anyof "cut,imp"
```

This bonus now allows the +5 to the owner's armor divisor to be applied for any mode where the damage type contains any occurrence of cut or imp.

NOTE: many combinations of COMPARISON with SUBCRITERIA will make no sense, or will be incredibly confusing to figure out, especially if they're negative in connotation, such as EXCLUDES with NONEOF. I encourage you to always use positives when possible.

Compound Bonuses

Compound bonuses are bonuses that use a single bonus listing to grant a bonus to one or more possible targets, but that will **not** add the bonus more than once to any particular target, no matter how many of the listed targets it may qualify as.

For example, gives(+1 to (CO:Air, CO:Water)) is a compound bonus, as it grants a bonus to both the colleges of Air and Water, but any spell that might belong to both colleges will still only get a single +1 bonus, not a +2 for belonging to both colleges. Note that if the GIVES() had been written gives(+1 to CO:Air, +1 to CO:Water), a spell belonging to both colleges **would** receive a +2, because it would satisfy both targets for both bonuses—this is what compound bonuses are meant to address.

Note that compound bonuses do not work with target tag bonuses, or with targets of ME:: or OWNER::. To create a compound bonus, the structure is the same as any other GIVES() bonus, except that the targets should be enclosed in parentheses as a comma separated list. If a target name includes commas, it may be enclosed in quotes or curly braces.

Target Tag Bonuses

It's possible to target bonuses to a number of specific trait tags. To do so, you use the same TRAIT::TAGNAME format that you use to access tag values in other areas. However, because GCA doesn't support granting bonuses to every possible tag, only some tags are valid targets. These are all valid target tags: ACC, ARMORDIVISOR, BLOCKAT, BREAK, DAMAGE, DAMTYPE, DB, DR, EFFECTIVEST, LEVEL, MINST, PARRY, PARRYAT, PARRYSCORE, POINTS, RADIUS, RAISERULEOF, RANGEHALFDAM, RANGEMAX, RCL, REACH, ROF, SHOTS, and SKILLSCORE (note that LEVEL and POINTS are redundant with the normal bonus handling).

When GCA calculates values for many of these tags, it stores the results of the calculation in new tags, which use the same name but with a CHAR prefix. For example, GCA will take the original ACC() tag, evaluate it along with all bonuses targeted to::ACC for the trait, and then place the final result in the CHARACC() tag. (You may think of this as the original tags containing the default values, and the CHARX versions containing the character specific values.)

Because GCA is evaluating the targeted tags along with the bonuses, it may be possible to target them with text that adjusts the effective starting values before the calculations, or to adjust otherwise text-only tag values. You may target the text value of a tag by including a $ sign at the end of the target tag, like so: TRAIT::TAGNAME$. For example, to apply a text bonus to the DAMTYPE() tag of a trait, you might use an expression like gives(=" dbk" to owner::damtype$), which would append the text to the end of the existing DAMTYPE() tag data.

Special Case $ Bonuses

There are several special case bonuses that can be applied to $ target tags.

NOBASE	clears the base value data before evaluating the rest of the bonuses, allowing for the entire CHARX value to be determined by granted bonuses.

NOCALC	prevents the tag from being calculated, if it is normally calculated, and results in the CHARX tag being just the concatenated base and text bonus values.
NOSIZE	only available for bonuses targeted to REACH$, this prevents the application of adjustments based on the character's Size Modifier.

Calculation Methods

Because different tags contain different data, and have different needs, there are many different ways they may be calculated, which means applying a particular bonus to one may not result in the same value as applying that bonus to another.

The table below shows the various target tags, and where the final value is stored, if damage modes are supported, and the general method used to arrive at the final value. All calculations begin with the text value of the given target tag.

Target Tag	Stored To	Modes	Method Used
ACC	CHARACC	X	VALUEMETHODSUFFIX
ARMORDIVISOR	CHARARMORDIVISOR	X	VALUEMETHOD
BLOCKAT	BLOCKLEVEL		SCOREMETHOD
BREAK	CHARBREAK	X	VALUEMETHOD
BULK	CHARBULK	X	VALUEMETHOD
DAMAGE	CHARDAMAGE	X	Special.
DAMTYPE	CHARDAMTYPE	X	Special. Do NOBASE, Append Bonus String, Damage Mode Special Case Subs, Text Function Solver
DB	CHARDB		Simple math; no text bonuses supported
DEFLECT	CHARDEFLECT		VALUEMETHOD
DR	CHARDR		Simple math; no text bonuses supported, suffix preserved
EFFECTIVEST	CHAREFFECTIVEST	X	Special. ST=DamageBasedOn, Do NOBASE, Do NOCALC, Append Bonus String, Solver, Apply Bonuses
FENCINGPENALTY	CHARFENCINGPENALTY		VALUEMETHOD
FORTIFY	CHARFORTIFY		VALUEMETHOD
LEVEL	LEVEL		Special.
LOCATION	CHARLOCATION		Special. Text and text functions only.
MINST	CHARMINST	X	VALUEMETHODSUFFIX
PARRY	CHARPARRY	X	Special. Do NOBASE, Append Bonus String, Damage Mode Special Case Subs, TextFunctionSolver, "no", U/F suffix perserved, Solver, Apply Bonuses
PARRYAT	PARRYLEVEL, PARRYATBONUS, PARRYATMULT		SCOREMETHOD
PARRYSCORE	CHARPARRYSCORE	X	Special. There is no base PARRYSCORE() tag; bonuses targeted to it are used in the calculation of CHARPARRYSCORE() based on CHARSKILLUSED(), CHARPARRY(), PARRYAT(), PARRYATBONUS(), and PARRYATMULT().

Target Tag	**Stored To**	**Modes**	**Method Used**
POINTS	POINTS		Special.
RADIUS	CHARRADIUS	X	RANGEMETHOD
RAISERULEOF	RAISERULEOF		Special. There is no base RAISERULEOF() tag; bonuses targeted to it are stored in RAISERULEOF(); only +/- bonuses are supported
RANGEHALFDAM	CHARRANGEHALFDAM	X	RANGEMETHOD
RANGEMAX	CHARRANGEMAX	X	RANGEMETHOD
REACH	CHARREACH	X	REACHMETHOD
SHOTS	CHARSHOTS	X	VALUEMETHODSUFFIX
SKILLSCORE	CHARSKILLSCORE	X	Special. There is no base SKILLSCORE() tag; bonuses targeted to it are used in the calculation of CHARSKILLSCORE() when evaluating SKILLUSED()

Most of the tags are calculated using one of these methods, which describe here the general process used to find the final stored value.

RANGEMETHOD Do NOBASE, Do NOCALC (Append Bonus String, Exit), Preserve Known Suffixes, DamageBasedOn/ST adjustment, Append Bonus String, Solver, Apply Bonuses, Apply Suffix

REACHMETHOD Text Function Solver, Do NOBASE, Do NOCALC (Append Bonus String, exit), Do NOSIZEMOD, Append Bonus String, Damage Mode Special Case Subs, Text Function Solver, Do ReachBasedOn, Adjust for Size Modifier

SCOREMETHOD Do NOBASE, Append Bonus String, Damage Mode Special Case Subs, Text Function Solver, Solver, Apply Bonuses

VALUEMETHOD Do NOBASE, Append Bonus String, Damage Mode Special Case Subs, Text Function Solver, Solver, Apply Bonuses, Preserve Empty

VALUEMETHODSUFFIX Do NOBASE, Preserve Suffix, Append Bonus String, Solver, Apply Bonuses, Apply Suffix, Preserve Empty

Special Case Equipment Targets

There are a few special targets for use with equipment items, as well.

Two of these targets are BASECOST and BASEWEIGHT, which allow for targeting bonuses to the base cost and weight of the equipment item, before any child items are included.

Two other targets are CHILDRENCOSTS and CHILDRENWEIGHTS, which allow for targeting bonuses to the total combined cost and weight of all child items, before they are included in the total cost and weight of the equipment item.

The final two targets are just COST and WEIGHT, which apply the bonuses to the final value, including the total costs and weights of any child items.

Exceptions to Bonuses

Bonuses now support the ability to declare an exception to the bonus, so that items can exclude themselves from receiving the bonus if they fall under the exception's parameters.

> GIVES([=] BONUS TO TRAIT [UPTO LIMIT] [ONLYIF TARGET[::TAG] = Y] [UNLESS TARGET[::TAG] = Z] [WHEN "CIRCUMSTANCE"] [LISTAS "FROM BONUS TEXT"])

Pay attention to the order of the GIVES() structure, as the order of the various key words and their data is very important.

There are two ways to create a bonus exception:

1) Unless

Create the exception using the new UNLESS keyword:

> UNLESS TARGET[::TAG] = Z

Note that the TARGET keyword is also required if you're looking to create the exception based on the value of one of the bonus receiver's tags. The UNLESS block should solve down to a True or False value, much like the If part of the @IF() function. The UNLESS clause is fully Solver enabled.

As an example, the bonus for Jack of All Trades might now be written like this:

> gives(+1 to skills unless target::points > 0)

(Note that I don't know off the top of my head if the Jack of All Trades bonus is supposed to apply to all defaults, like the bonus here, or only to defaults from attributes, in which case this example would need additional work.)

2) OnlyIf

This is basically the same as the UNLESS clause, but may be easier for some folks to visualize, because it's positive instead of negative. ONLYIF looks like this:

> ONLYIF TARGET[::TAG] = Y

Note that the TARGET keyword is also required if you're looking to create the exception based on the value of one of the bonus receiver's tags. The ONLYIF block should solve down to a True or False value, much like the If part of the @IF() function. The ONLYIF clause is fully Solver enabled.

ONLYIF and UNLESS both restrict when a bonus is applied. You may use either or both, but if you use both in the same bonus, you must keep in mind that ONLYIF will be checked first, and if the target **does not** satisfy the requirement, then the bonus **will not** be applied, regardless of the UNLESS clause. The UNLESS clause further restricts bonus application to traits that satisfy the ONLYIF requirement; it **never** restricts or excepts the results of the ONLYIF clause. If you only use one or the other clause, then when the exception or requirement applies should be fairly clear.

Targets

Bonuses can now be granted TO CULTURES or TO LANGUAGES.

Bonuses can now be granted to a Cultural Familiarity or Language based on their category using

> TO CUCAT: CATEGORY

or

> TO LACAT:CATEGORY

See also the [Bonus Targets](#) block in the [General Information](#) section.

User Targetable Bonuses (%ChosenTarget%)

Support has been added for "user targetable" bonuses, which will allow the user to choose the targets to which the bonus is meant to apply.

Items that grant this new type of bonus will use %CHOSENTARGET% as the Target part of the bonus, such as

> gives(+1 to %chosentarget%)

GCA will manage the new CHOSENTARGETS() tag, which will track the names of the target items, and provide a UI for it through the use of a new button on the Edit Traits dialog, which will pop up a pick list from which they can choose target items. When bonuses are generated by the trait, GCA will create one bonus for each target, in each case replacing the %CHOSENTARGET% with the name of the target item.

File authors can limit the available targets presented to the user by using the TARGETLISTINCLUDES() tag on the item; see the related info in the Tag Detail Information section.

Gains Bonuses (the From Keyword)

Bonuses can now be gained from another trait, rather than having to edit that other trait to have it give the bonus back.

> GIVES([=] BONUS TO ME[::TAGNAME] FROM TRAIT [UPTO LIMIT] [ONLYIF TARGET[::TAG] = Y] [UNLESS TARGET[::TAG] = Z] [WHEN "CIRCUMSTANCE"] [LISTAS "FROM BONUS TEXT"])

Note that this type of bonus still comes from the GIVES() tag, but requires the additional use of the FROM keyword. When using this construct, the TO ME phrase is also required.

For example, the tag

> gives(+1 to me from AD:Magery)

would apply a bonus of +1 per level of the Magery advantage to the trait containing the tag.

The 'gains' feature should support the expanded keywords such as ONLYIF and UPTO. To target a specific tag of ME, do that normally using ME::TAGNAME.

Gains bonuses can support gaining bonuses from the tags of FROM references, such as points, level, or whatever. For example:

> gives(+1 to me from char::example)

would give a bonus equal to the value of the example tag on the character (or 1 if it is not a numeric tag value, while a CHAR:: tag that is empty or doesn't exist would result in a bonus of 0).

You can also use this with traits, such as

> gives(+1 to me from SK:Acrobatics::points)

which would give a bonus of +1 per point spent in the Acrobatics skill to the trait. Same for using a 'from parent' construction, such as

```
gives(+1 to me from parent::points)
```

which would give +1 per point spent on the parent trait.

The ‘gains’ feature applies only to traits, not modifiers.

Target Tag Bonuses

Traits may now have the LOCATION() tag targeted.

GCA will currently only create a CHARLOCATION() tag if there are bonuses being applied to LOCATION(). However, this does allow us to make the Partial: Location modifiers for Damage Resistance more functional, because now their gives could be changed to something like this one for Partial: Arms:

```
gives(=-Owner::Level to DR, =+owner::level to owner::dr, =arms+nobase to owner::location$)
```

which will allow GCA to correctly apply the location-specific DR to the correct locations on the protection image and will allow the Protection window to display these limited forms in the *From Other Sources* listing.

Bonus Classes

See also Bonus Classes.

To include a bonus in a particular BonusClass, just use the CLASS keyword and specify the name or names of the classes to which it should belong after that (in quotes/braces if needed) within the bonus text. It might look like this:

```
gives(+1 to "SK:Acrobatics" upto 4 class "Talents, Acrobatics")
```

That would include the bonus as a member of the classes "Talents" and "Acrobatics".

Enhanced Parsing Changes

Using updated parsing routines, we can now allow for any of the different clauses to be used in any order within the text; just the actual bonus part needs to be listed first, as before. For example, you could now use

```
gives(+2 to "SK:Acrobatics" when "on Earth" listas "native gravity bonus" upto 4 onlyif target::points > 0)
```

or you could use

```
gives(+2 to "SK:Acrobatics" onlyif target::points > 0 listas "native gravity bonus" upto 4 when "on Earth")
```

and either should work correctly and without issue.

Spaces around the keywords aren't required any more, except for around the TO and FROM keywords in the basic bonus block, but you must enclose other blocks in quotes or braces if they might include any of the other keywords: ONLYIF, UNLESS, LISTAS, CLASS, UPTO, and WHEN. GCA will strip the containers after parsing. (Parsing should now honor braces as well as quotes as containers for these clauses.)

Group()

Applies to: traits and modifiers

The GROUP() tag allows for including an item in a Group without having to adjust the Groups listing in the data file—in fact, a corresponding Groups listing is not required, so completely off the cuff groups could be created and handled strictly through the use of the GROUP() tag. Any NEEDS() or GIVES() that affects a GR:GROUP will affect any item that includes the specified GR:GROUP in the GROUP() tag.

The GROUP() tag works just like a CAT() tag, in that it's a list of group names separated by commas. The GR: prefix should **not** be used inside the GROUP() tag.

For example, a gives(+2 to GR:Alpha) would affect any trait that had a group(Alpha) tag, as well as any item listed in any existing Alpha group listing in a data file.

Modifiers

For modifiers, GROUP() acts as both a CAT() and a part of the internal name of the modifier. A modifier can only belong to a single GROUP(), and if an otherwise identical modifier is found in two different modifier groups, that is two completely different modifiers as far as GCA is concerned. This usage of GROUP() is therefore very different from that of traits.

The group for a modifier is set by the <GROUP> header in the data file. If the modifier in the data also has a GROUP() tag, GCA will consider this an error: the group set by the header will not be overridden, and an error will be logged if Verbose logging is on. This is not considered a critical error, and for ease of use when copying-and-pasting modifiers into modifier definitions elsewhere in a file, it may help to have a GROUP() tag specified. However, you should be aware of this behavior.

Hide()

Applies to: traits

The HIDE() tag is a flag tag that would cause the marked trait to be hidden on the character by default. (You could always use Show Hidden to see hidden traits, so being hidden is more organizational than sneaky.)

The components of most racial templates and meta-traits are hidden by default.

HideMe()

Applies to: traits

Allows you to use a formula to set the value of the HIDE() tag.

The Solver is used to return a numeric result. If zero, HIDE() is removed, otherwise HIDE(YES) is added to the trait.

Hides()

Applies to: templates

This tag specifies that a trait should hide all the component traits added or created by this trait. In practice, this means that all the added or created traits will receive the HIDE(YES) tag, which tells GCA not to display the trait to the user. HIDES() may be used on templates, or on any trait using ADDS() or CREATES(), but you should not use the HIDE() tag yourself.

This tag is a flag tag.

Highlight()

Applies to: traits

The HIGHLIGHT() tag used to be a flag tag that would cause the marked trait to be highlighted with a yellow background.

GCA now supports any predefined color as an option for the HIGHLIGHT() tag. GCA will use the name of the color to derive the highlight that is drawn, with a transparency value of 100 (out of 255). The predefined colors that can be used are shown here <https://docs.microsoft.com/en-us/dotnet/api/system.windows.media.colors?view=net-5.0>.

If the tag is HIGHLIGHT(YES), GCA will still pick yellow as the highlight color, but GCA has been updated to allow setting the HIGHLIGHT() to YELLOW, PINK, MAGENTA, LIME, CYAN, RED, GREEN, or BLUE.

HighlightMe()

Applies to: traits

Allows you to use a text formula to set the value of the HIGHLIGHT() tag.

The TextFunctionSolver is used to return a text result. If the result is empty ("", nothing), the HIGHLIGHT() tag is removed, otherwise the result is placed into the HIGHLIGHT() tag.

HitTable()

Applies to: templates

This tag allows you to specify a hit table by name, which will be used to define the default hit location table used by the character. This should refer to a hit location table found in the library, or it will fail to change the table.

Ident()

Applies to: traits

Used in conjunction with COUNTASNEED(), the IDENT() tag allows for identifying traits that may include as prereqs some traits that are normally not allowed as such. See COUNTASNEED() for details.

Init()

Applies to: traits other than skills, spells, and equipment

This tag allows you to specify the initial level of the trait when first added to the character. This tag should only be used in standard trait definitions, as it may not work correctly when used in a definition that is part of a CREATES() tag.

InitMods()

Applies to: traits and modifiers

This tag allows you to specify any modifiers that should be applied to the trait when it is first added to the character. This tag should only be used in standard trait definitions, as it may not work correctly when used in a definition that is part of a CREATES() tag.

You may specify multiple modifier definitions, but they must be separated by pipe (|) characters, since the definitions use many commas. You may enclose each definition in quotes or braces if desired.

Example:

```
initmods(_
        Reliability: Somewhat Reliable, *1,
                shortname(Somewhat Reliable), group(Reliability) _
        | Frequency: roll of  9 or less (Fairly often), *1,
                shortname(9 or less), group(Frequency of Appearance) _
        )
```

If you want the initial modifier to be a particular level of the modifier, you may include the LEVEL() tag in the definition.

InPlayMult()

Applies to: attributes

Not used in GURPS 4th Edition.

When the Character In Play check box is checked, the current levels and costs of attributes are “frozen”, and any increment above the “frozen” level costs the normal level cost, multiplied by this value. In GURPS 3rd Edition, for example, inplaymult(2) would have been used, and once the character was in play, attributes cost double to raise.

Invisible()

Applies to: system traits

Added support for the SystemTrait only tag INVISIBLE(YES), which is a flag tag telling GCA that the item is 'invisible' and should never be shown to the user.

```
invisible(yes)
```

Support for this is added with the understanding that it simplifies certain types of data constructs, allowing an 'invisible' trait to be added to the character by another trait, whereupon it will be visible, but presumably as a child trait, or perhaps otherwise linked to the adding item.

For example, you could create Equipment versions of certain advantages by duplicating the advantages with "Gear" name extensions and zeroed costs, but make them invisible so they aren't taken accidentally (or make invisible the current ones if they're only available as gear in your game). Then you could make Equipment items that add those invisible traits to get the appropriate effects--the added traits are visible to the user as normal traits.

Note: GCA does not support this tag for attributes. Also note that categories for these traits are still considered valid, so a category filled with invisible items will appear like an empty category to the user, which may be confusing in some cases.

ItemNotes()

Applies to: traits

The ITEMNOTES() tag provides a mode-enabled way of specifying notes for traits. Usually for weapons or attacks, but not required as such. It will create a mode structure even if no other mode-enabled tags are used.

```
itemnotes(_
        {} | {May get stuck; see Picks (p. B405).} | {}_
)
```

Each note should be an individual item, and multiple notes should be separated by commas. Enclose notes in braces or quotes if necessary to avoid incorrect parsing. Notes for each different mode should be separated by pipe | symbols.

Mode-enabled.

LC()

Applies to: traits with damage modes

The LC value of the weapon or attack.

This tag is mode-specific.

Level()

Applies to: modifiers

This tag allows you to specify the level at which the modifier should be applied. This is generally only used when the modifier is being defined as part of something else, such as being added to a trait being added through the ADDS() tag.

LevelNames()

Applies to: traits other than skills, spells, and equipment

This tag allows you to specify names for the levels of the trait, rather than simply using numeric values.

The level names should be a comma-separated list, using quotes around names that may include a comma.

You may also specify the zero-level name, by enclosing it in square brackets as the first name in the list of names. You may not enclose it in quotes, so do not use any restricted characters.

Example:

```
Appearance, -4/-8/-16/-20/-24, mods(Appearance), upto(5), page(B21),
    levelnames([Average], Unattractive, Ugly, Hideous, Monstrous, Horrific),
    page(B21), taboo(AD:Appearance)
```

The LEVELNAMES() in this example includes Average for the level 0 value of the trait.

Loadout()

Applies to: equipment

This is a trigger tag, which is activated and processed when a trait is added to a character.

```
LOADOUT( LOADOUT [, LOADOUT2 [, ... ] ] )
```

When used, the item is automatically added to the loadouts specified. If the loadouts don't exist, they'll be created. Separate additional loadouts with commas, and use quotes or braces as required.

For example

```
loadout(Crime, Punishment)
```

will add the item to both the Crime and Punishment loadouts when added to the character.

GCA will now automatically apply protection from any armor or shield items so added. This seemed to be the most likely desired action, so it was made the default. If you do NOT want that to happen, include #NOARMOR as part of the loadout name inside the #LOADOUT() directive for each loadout to not receive protection.

If you are adding a shield, you can use #ARC() as part of the loadout name to specify the protected arc. At this time, the supported arcs are none (the default), back, left arm, and right arm.

See the Adds() tag **Expanded Features** section for more information, as #LOADOUT() supports the same features.

Note that supporting an empty LOADOUT() tag to add protection to 'All unassigned items' means that using an empty LOADOUT() tag on a trait will trigger GCA trying to assign it to a loadout and to apply protection, so don't include empty LOADOUT() tags unless that's what you want to happen.

Links()

Applies to: traits

This works similar to OWNS() but creates a different relationship, where two traits are linked together for ease of referencing certain features. The relationship is instantiated during an ADDS() or CREATES() using LINKS(YES) on the source trait, which is the one doing the adding or creating.

This process will bond the relationship using these tags: LINKED(IDKEY) and LINKEDFROM(IDKEY).

The LINKED(IDKEY) tag on the target trait (the one that was added or created) will get the IDKEY of the source trait. It is assumed that this target trait depends on the source trait for some data that it needs.

The LINKEDFROM(IDKEY) tag will be added to the source trait (the one that did the adding or creating), so that it will know that it's being linked to from another trait and can update that trait when it changes.

LINKEDFROM() can have many different IDKEYS in a comma-separated list, but LINKED() can be only a single IDKEY value.

The destination trait can then depend on having current data, and can refer back to the LINKED() trait in formulas using LINKED:: much as you can refer to parents with PARENT:: or modifier owners with OWNER::.

Here’s a complete example from Payload in Basic Set:

```
* The Payload Advantage here doesn't need or use a weightcapacity() because it doesn't
* actually carry anything, so we just calculate it in variables for display purposes.
* The creates() makes the Equipment parent item that has the actual weightcapacity() that we
* care about for actual use.
Payload, 1/2,
mods(Payload),
page(B74),
cat(Exotic, Physical),
vars(_
	%cap% = "ST:Basic Lift"/10 * %level,
	%units% = $if( ST:Metric = 1 THEN kg ELSE lbs ),
	%weight% = Max: $eval(%cap%) %units%,
	%nameext% = $val(me::nameext),
	%ext% = $if( @len(%nameext%) = 0  then "(%weight%)" else "(%nameext%; %weight%)" )_
	),
displaynameformula( $val(me::name) $val(me::level) %ext% ),
links(yes),
creates(_
	{EQ:Payload Items,
	weightcapacity( "ST:Basic Lift"/10 * linked::level),
	vars(_
		%units% = $if( ST:Metric = 1 THEN kg ELSE lbs ),
		%weight% = Max: $eval(me::weightcapacity) %units%,
		%nameext% = $val(me::nameext),
                %ext% = $if( @len(%nameext%) = 0  then "(%weight%)" else "(%nameext%; %weight%)" )_
                ),
        isparent(yes),
        displaynameformula( $val(linked::fullname) Items %ext% ) }_
        )
```

When the user takes the Payload advantage, it also creates a Payload Items equipment item that is linked to it. Payload Items is a parent item so that the user can add children to it, which will use up the available capacity. The LINKED() relationship allows it to be updated with the correct capacity information as the Payload advantage is adjusted.

You cannot have LINKS(YES) and OWNS(YES) on the same source trait; OWNS() will take precedence and LINKS() will not be applied.

GCA will also create these related supporting tags: LINKEDNAME() and LINKEDFROMNAMES(), which are the names of the traits related to the IDKEYS in the LINKED() and LINKEDFROM() tags, respectively; and BROKENLINKS() which contains the IDKEYS from LINKED() or LINKEDFROM() that can no longer be found.

Location()

Applies to: equipment

LOCATION() specifies the areas covered by armor items, and is used to determine what body parts receive protection values from the armor equipped in each loadout.

What is new in GCA5 is how this tag works in conjunction with protection values.

If multiple locations are specified for a piece of armor, locations specified later in the list are supposed to apply more specific armor values. For example, location(full suit, neck) and dr(4/2) should apply 4/2 to all locations covered by full suit and by neck, even though neck is covered by full suit and the DR is the same, because no more specific value was provided. If, instead, that DR() tag was dr(4/2, 1), then locations covered by full suit should all get the 4/2 value, while neck should only receive the 1 value.

Locks()

Applies to: templates

This tag specifies that a trait should lock all the component traits added or created by this trait. In practice, this means that all the added or created traits will receive the LOCKED(YES) tag, which tells GCA not to allow the user to modify the trait. LOCKS() may be used on templates, or on any trait using ADDS() or CREATES(), but you should not use the LOCKED() tag yourself.

This tag is a flag tag.

Lockstep()

Applies to: templates

Using the LOCKSTEP(YES) tag specifies that GCA should try to require that component advantages and disadvantages have the same level as the template. A template must have a multiple cost in the standard format, even if it is 0/0 or similar, in order to be leveled.

This tag is a flag tag.

MainWin()

Applies to: attributes

This tag specifies the order in which to display the various attributes in the main display area for attributes. The MAINWIN() tag should contain a simple numeric value specifying the display location. For example, mainwin(1) would be used on the trait that should be displayed first, and mainwin(5) would be used on the trait that should be displayed fifth.

MaxDam()

Applies to: traits with damage modes

This tag specifies the maximum damage, as a standard GURPS damage notation (such as 3d+2), that the damage mode may have.

MaxScore()

Applies to: attributes

This tag specifies the maximum allowable value for the attribute. This tag is math enabled.

> MAXSCORE(VALUE [LIMITINGTOTAL])

If the optional keyword LIMITINGTOTAL is used, GCA will attempt to limit the total score of the attribute to VALUE, including any bonuses that may be applied. Normally, the MAXSCORE() would be limited before any bonuses are applied, allowing for a final score above VALUE.

MergeTags()

Applies to: traits

This tag allows you to add or insert tags to existing traits when the current trait is added to the character.

> MERGETAGS(IN [ALL] "TRAIT" with "TAG LIST" [#PRESERVE] [, IN [ALL] {TRAIT} with {TAG LIST}])

TRAIT	is the trait being targeted for the tag merge, using the name and appropriate prefix tag. If the TRAIT name includes any of the keywords or operators, you should enclose it in quotes or braces. You may also target the tags of the character using the CHAR keyword as the TRAIT target.
TAG LIST	is the list of tags to be merged into the target trait. The TAG LIST should generally be enclosed in quotes or braces; use braces if any tag values include quotes.

If the ALL keyword is used, then the TRAIT being targeted is intended to affect all traits with the same base name.

You may not target multiple traits as part of the TRAIT, except by use of ALL. This tag allows for only a single target per IN block, although you may include multiple IN blocks, separated by commas.

Example:

> MergeTags(in "SK:Forward Observer" with "cat(Military), page(CI151)", in "SK:Darts" with "cat(Hobby), page(CI146)")

Here, SK:Forward Observer and SK:Darts are both targeted for additional category and page numbers.

Example 2:

> mergetags(in char with "race(vampire)")

Here, the character has vampire merged into its RACE() tag.

#PRESERVE

GCA has very few 'undo' features, but this is one place where there is some of that, and if you do a REPLACETAGS() or MERGETAGS() and then remove the template that performed it, GCA will undo that tag change. You can use #PRESERVE to make the change stick. Insert the command into the 'with' block somewhere and GCA will strip it out when found, but that will tell GCA not to create a changelog entry for it.

For example

> replacetags(in char with "Profession(Bandit)")

will change (or add) a Profession tag to the character, with a value of Bandit, but removing the template that made that change will remove the tag value again (or change it back to what it was). On the other hand, using

replacetags(in char with "Profession(Bandit)"#preserve)

will ensure that the profession remains, even if that template is removed.

MinScore()

Applies to: attributes

This tag specifies the minimum allowable value for the attribute. This tag is math enabled.

Message()

Applies to: templates

This tag works like the #MESSAGE directive, but in a tag.

Works in TRIGGERS() or is processed in the standard tag list last, just before cleaning up directives, before item calculations begin on newly created items.

This is primarily meant for use in TRIGGERS() since the tag can only handle a single message, but it works outside of TRIGGERS() if you just need one message at the end.

MinST()

Applies to: traits with damage modes

This tag specifies the minimum ST score required for using the weapon or attack without penalties. This should be a simple numeric value, possibly with simple suffix text.

This tag is mode-specific.

MinSTBasedOn()

Applies to: traits with damage modes

If this tag exists, GCA will use the stat specified within instead of Striking ST to do the MinST check. If the stat specified within can't be found, GCA will instead use the default Striking ST (or ST, if none) to do the check.

This tag is mode-specific.

Mitigator()

Applies to: modifiers

The MITIGATOR() tag allows for specifying that the modifier will mitigate the target trait. In practice, this means that the MITIGATOR() tag will result in a similar MITIGATED() tag being added to the trait.

There can be several states:

- MITIGATOR(YES) will set MITIGATED(YES), which means bonuses will be changed to conditional bonuses, and taboos will be ignored.
- MITIGATOR(TABOO) will set MITIGATED(TABOO), which means taboos will be ignored, but bonuses will remain bonuses.
- MITIGATOR(GIVES) will set MITIGATED(GIVES), which means bonuses will become conditionals, but taboos will remain.

Mode()

Applies to: traits with damage modes

This tag specifies the name of the mode. This should be the simplest possible text name for the mode, that does not include restricted characters.

This tag is mode-specific.

Mods()

Applies to: traits and modifiers

This tag specifies a list of modifier groups that are applicable to the trait or modifier.

Name()

Applies to: traits and modifiers

This specifies the name of the trait or modifier. This is almost never used, as the name is a non-tagged part of the trait and modifier definitions in the data file.

NameExt()

Applies to: traits and modifiers

This specifies the name extension of the trait or modifier. This is almost never used, as the name is a non-tagged part of the trait and modifier definitions in the data file.

Needs()

Applies to: traits

This tag allows for specifying the prerequisites of the trait.

NEEDS() blocks can get complicated, so be aware that GCA parses needs blocks based on OR (|) markers first, if any are present, and then processes the AND (,) sections within them.

If you want to include a simple OR section within an AND section, you should enclose the OR items in parenthesis, so that they won't be parsed prematurely.

The simplest NEEDS() block consists only of items separated by commas; in this case, all specified items are required. For example:

```
needs(SK:Diplomacy, SK:Dancing)
```

which requires both skills.

A slightly more complex block is a simple AND, including a simple OR:

```
needs(AD:Possession, (AD:Ally | DI:Dependent))
```

This block needs AD:Possession *and either* AD:Ally *or* DI:Dependent.

If the OR block was not included in the parens, GCA would have parsed on it first, and the meaning would have changed, as then having DI:Dependent alone would have satisfied the need. That is similar to the idea of this tag, with three OR blocks:

```
needs(SK:Diplomacy, SK:Dancing | SK:Intimidation, SK:Dancing | SK:Leadership, SK:Dancing)
```

In this case, SK:Dancing is required in each OR block, but it along with SK:Diplomacy, SK:Intimidation, *or* SK:Leadership would satisfy the needs.

You may specify a particular level or number of points needed, using a comparison operator, like so:

```
needs(SK:Computer Programming = 1pts)
```

GCA will recognize >,<,=, >=, <=, and == as valid comparisons. Note that when using =, GCA understands that to mean any higher value is also okay, just as if you'd used >=. If only a specific value is allowed, you must use ==.

By default, if you do not use a comparison, GCA will require at least one point spent in any specified Skills or Spells, or at least one level of any other trait, in order for a trait to be considered as possibly satisfying a need.

This tag is math enabled, but generally only on the right side of any comparison operator.

This tag is also string function enabled, and will process string functions first, if detected, so that you may return different needs based on different circumstances, such as different levels of the trait.

Special Cases

Needs checking supports many special cases to allow for a broader variety of prerequisite constructs.

X SPELLS FROM Y [OTHER] [COLLEGES | CATEGORIES | CATS]

Some number of spells from some number of colleges is required. If the OTHER keyword is used, then the college that the containing trait belongs to is not counted among the colleges required.

Any one of the optional keywords COLLEGES, CATEGORIES, or CATS may be used, as all mean the same thing to GCA when the containing trait is a spell.

X SKILLS FROM Y [OTHER] [CLASSES | CATEGORIES | CATS | SKILL CLASSES | SKILL CATEGORIES | SKILL CATS]

As above, but for skills.

X [QUIRKS | PERKS | SPELLS | SKILLS]

Some number of traits of the specified type is required.

X [OTHER] [COLLEGES | SPELL CATEGORIES | SPELL CATS]

Some number of colleges is required. If the OTHER keyword is used, then the college that the containing trait belongs to is not counted among the colleges required.

Any one of the optional keywords COLLEGES, SPELL CATEGORIES, or SPELL CATS may be used, as all mean the same thing to GCA.

X [OTHER] [CLASSES | SKILL CATS | SKILL CATEGORIES]

As above, but for skill categories.

X CATEGORYNAME [= VALUE]

Some number of traits is required, from the category CATEGORYNAME, at the optionally specified VALUE level or points.

For example, needs(3 CO:Enchantment=10) specifies that 3 spells are needed from the Enchantment college, and each must be at least level 10.

[X] GR:GROUPNAME [= Y]

Some number of traits is needed from a specified group. X is optional and specifies the number of items from the group that are needed at the Y level. If X is not specified, then **all** items from the group are needed. Y is the level at which the items are needed

NewMode()

Applies to: traits

This tag provides a simpler way to handle weapon modes in the data files. The NEWMODE() tag is used for every mode desired, and each contains the information for a single mode. It looks something like this:

```
Hatchet, techlvl(0), break(0), lc(4), basecost(40), baseweight(2), page(B271, B276),
        mods(Equipment, Melee Quality, Cutting Class Quality), calcrange(yes),
        newmode(Swing, damage(sw), damtype(cut), reach(1), parry(0), minst(8), notes([1]),
                skillused(Axe/Mace, DX-5, Flail-4, Two-Handed Axe/Mace-3)),
        newmode(Thrown, damage(sw), damtype(cut), acc(1), rangehalfdam(ST*1.5),
                 rangemax(ST*2.5), rof(1), shots(T(1)), minst(8), bulk(-2),
                skillused(Thrown Weapon (Axe/Mace), DX-4))
```

The information inside the NEWMODE() tag is formatted like a standard weapon item (first item is name of the mode, followed by a listing of tags), but just for items pertinent to that mode. If you define any tag in one mode, you need to be sure to also define it in any other mode that uses the same information, even if the values are the same. For example, in the example above, if another mode, say two handed throw was created, you'd have to include the SHOTS() and ROF() tags, for example, even though they'd likely be exactly the same. (When defining modes the old way, you could just leave off the ending mode values if they were the same as the previous ones, and GCA would use the last provided value. However, in this case, each time a new mode is added on, GCA separates all existing mode information from new info with the | separator, because it has no way of knowing if any subsequent modes will add additional info. It's a tradeoff in easier creating of modes against having to include additional duplicate data.)

Because of the way GCA handles modes, you **always** have to define a DAMAGE() tag for each mode. If you don't, GCA won't correctly process the mode information, and it will appear to be missing on the character sheet.

NEWMODE() is the **only** tag allowed to be used more than once per trait definition.

Notes()

Applies to: traits

This includes notes information from the source material.

Note that NOTES() was originally defined and meant for use with the numeric reference indexes on weapon tables and the like, so that information was meant to fit in a narrow column of one-half inch or so in width. Many data file authors didn't realize that, however, and used NOTES() for general trait notes. Please use USERNOTES() for general trait notes, and ITEMNOTES() if you want to add longer notes for particular modes.

For displaying very short info, such as the energy cost range of a spell, using NOTES() is a fine alternative that will then fit nicely in the Notes column of an attack table or a grimoire.

OptSpec()

Applies to: skills and spells

This tag is used to specify that a skill is an optional specialty. This is usually handled within GCA but may be set in the data files for specified items. Use optspec(1) to designate an optional specialty; only that value will work.

Owns()

Applies to: templates

This tag specifies that a trait should own all the component traits added or created by this trait. Costs of owned traits are included in the cost of the owning trait and are not included individually in the totals for their actual trait type. Owned traits with negative costs are also not counted against any disadvantage limit, since their costs are included in their parent; if the containing item has a negative cost, it will count against the disadvantage limit normally.

Normal usage is owns(yes).

This tag is a flag tag.

Page()

Applies to: traits and modifiers

This tag includes the page reference for the trait.

ParentOf()

Applies to: traits

This tag allows for specifying a list of items which are to be made children of the containing item when it is added to the character. Usually, you'll want to add those items with ADDS() or CREATES(), first. Note that PARENTOF() will steal the specified items from another parent, if they happen to already be assigned to one.

Parry()

Applies to: traits with damage modes

This tag allows you to specify the parry adjustment for the weapon or attack. This should be a simple numeric value, possibly with simple suffix text.

This tag is mode-specific.

ParryAt()

Applies to: traits

This tag allows you to specify the normal Parry score when using the trait to parry. For example

```
parryat(@int(%level/2)+3)
```

uses one-half of the value of the skill's level (dropping fractions), plus 3, for the trait's Parry.

This tag is math enabled.

Race()

Applies to: templates

This tag allows you to specify a race, which will be inserted into the character's Race field. This will replace any existing race that may already have been added to the character.

Radius()

Applies to: traits with damage modes

This tag specifies the radius of an attack or other effect. This should be a simple numeric value, or a simple suffix for miles or kilometers (GCA recognizes MILES, MI., and KM).

This tag is mode-specific.

RangeHalfDam()

Applies to: traits with damage modes

This tag specifies the half-damage range of an attack or other effect. This should be a simple numeric value, or a simple suffix for miles or kilometers (GCA recognizes MILES, MI., and KM).

This tag is mode-specific.

RangeMax()

Applies to: traits with damage modes

This tag specifies the half-damage range of an attack or other effect. This should be a simple numeric value, or a simple suffix for miles or kilometers (GCA recognizes MILES, MI., and KM).

This tag is mode-specific.

Rcl()

Applies to: traits with damage modes

This tag specifies the Rcl value of an attack or other effect. This should be a simple numeric value.

This tag is mode-specific.

Reach()

Applies to: traits with damage modes

This tag specifies the reach value of an attack or other effect. This should be a simple numeric value, or a simple range of values, as used in the weapon tables.

This tag is mode-specific.

ReachBasedOn()

Applies to: traits with damage modes

This tag allows you to specify what attribute the reach value for this trait is based on. There are several reach related attributes defined in the Basic Set file, or you may use 0 to specify that this reach is fixed.

This tag is mode-specific.

RemoveMods()

Applies to: traits

This tag allows you to specify modifiers that should be removed from other traits.

> REMOVEMODS(MODNAME FROM TARGETTRAIT [, MODNAME FROM TARGETTRAIT])

MODNAME is the name and extension (if any) of the modifier to be found and removed. The whole MODNAME can be enclosed in quotes or braces if necessary (such as when it includes the "to" keyword or a comma).

TARGETTRAIT is the name of the trait from which you want to remove the modifiers, including prefix tag, and name extension (if any). Enclose it in quotes or braces if necessary.

You can also specify multiple modifiers or multiple targets by separating them with commas, but if you do so, you need to enclose the whole block in braces or parens, to be sure that they aren't parsed out separately before the correct time.

Here is a nonsense example:

> removemods((Bar, Zub) from (SK:Some Skill, SK:Another Skill))

As you can see, you can remove multiple modifiers from multiple skills, in multiple blocks, if desired.

Removes()

Applies to: templates

Specifies a list of traits to be removed from the character, separated by commas. Prefix tags are required.

> REMOVES([ALL] TRAITNAME [, TRAITNAME] [, ...])

You may use the ALL keyword to specify that all variations of the skill should be removed, otherwise the name and name extension specified must match exactly. For example:

> removes(All SK:Guns, SK:Accounting, "SK:Animal Handling (Dogs)")

will remove every Guns skill, only the normal Accounting skill, and only the specialized Animal Handling (Dogs). Without the ALL keyword on SK:Guns no specialized Guns skill would be removed.

RemovesByTag()

Applies to: templates

Includes one or more criteria specifications (separated by commas) for removing traits from the character. Criteria is provided in this format:

> TYPE WHERE TAG [IS | ISNOT | INCLUDES | EXCLUDES | LISTINCLUDES | LISTEXCLUDES] TAGVALUE

This is the same format as that used by the #BUILDCHARITEMLIST directive, so see that for more information on how each of the comparison types works.

Note that it is very dangerous to use criteria that specifies a non-match, because that means any trait without the specified tag will likely not match and will be removed. REMOVESBYTAG() is intended to be used primarily in conjunction with tag values added during a SELECTX() process, for further customizing templates with lenses later on, but data file authors will do as they will.

ReplaceTags()

Applies to: traits

This tag allows you to replace the tags of existing traits when the current trait is added to the character.

> REPLACETAGS(IN [ALL] “TRAIT” with “TAG LIST” [#PRESERVE] [, IN [ALL] {TRAIT} with {TAG LIST}])

TRAIT	is the trait being targeted for the tag replacement, using the name and appropriate prefix tag. If the TRAIT name includes any of the keywords or operators, you should enclose it in quotes or braces. You may also target the tags of the character using the CHAR keyword as the TRAIT target.
TAG LIST	is the list of tags to be replaced on the target trait. The TAG LIST should generally be enclosed in quotes or braces; use braces if any tag values include quotes.

If the ALL keyword is used, then the TRAIT being targeted is intended to affect all traits with the same base name.

You may not target multiple traits as part of the TRAIT, except by use of ALL. This tag allows for only a single target per IN block, although you may include multiple IN blocks, separated by commas.

Example:

```
ReplaceTags( in "ST:ST" with "up(5/10), down(-5/-10)", in all "SK:Public Speaking" with "cat(Presence),
page(house)")
```

Here, ST is being given new costs, and all the SK:Public Speaking skills are being given a new category and page number.

Example 2:

```
replacetags( in char with "race(gremlin),laugh(cackle)" )
```

Here the character has its RACE() changed to gremlin and a new LAUGH() tag added with a value of cackle.

#PRESERVE

GCA has very few 'undo' features, but this is one place where there is some of that, and if you do a REPLACETAGS() or MERGETAGS() and then remove the template that performed it, GCA will undo that tag change. You can use #PRESERVE to make the change stick. Insert the command into the 'with' block somewhere and GCA will strip it out when found, but that will tell GCA not to create a changelog entry for it.

For example

```
replacetags(in char with "Profession(Bandit)")
```

will change (or add) a Profession tag to the character, with a value of Bandit, but removing the template that made that change will remove the tag value again (or change it back to what it was). On the other hand, using

```
replacetags(in char with "Profession(Bandit)"#preserve)
```

will ensure that the profession remains, even if that template is removed.

ROF()

Applies to: traits with damage modes

This tag specifies the ROF value of an attack or other effect. This should be a simple numeric value.

This tag is mode-specific.

Round()

Applies to: attributes, modifiers

This tag specifies a value that determines how the attribute score should be rounded off.

You can now specify UP or 1 for rounding up, DOWN or -1 for rounding down, and NO, 0, and NONE for no rounding. Note, however, that modifiers only actually support rounding up or down, so any usage they see that isn't DOWN or -1 is the same as UP or 1.

This tag is math enabled.

RoundLastOnly()

Applies to: traits

This is intended for traits that have fractional base values, which are supposed to be modified by modifiers, and should only be rounded up after all modifiers are finished, rather than GCA's normal method of rounding after each tier of modifiers. Use cautiously, as this overrides certain specialty modifiers that might normally request rounding down for some tier.

RoundMods()

Applies to: modifiers

If ROUNDMODS(NO) is included, then no rounding will be performed when calculating modifiers applied to this modifier. This is the only way to prevent rounding of fractional values if any modifiers to the modifier exist.

This is for certain specialty cases and is almost never used.

ScopeAcc()

Applies to: traits

The SCOPEACC() tag reflects the bonus to acc that derives from a scope or similar such aiming aid.

The CHARSCOPEACC() tag will hold the bonus after any bonuses to SCOPEACC have been applied.

The CHARSCOPEACC() value will now be used to create the suffix value for the final CHARACC() tag, so acc(2) and scopeacc(2) with a +1 to scopeacc bonus would result in charscopeacc(3) and characc(2+3).

Select()

Applies to: templates

This is the same as SELECTX(), but as used in TRIGGERS(). Since TRIGGERS() processes included tags in the order they're found, no index marker is required. See SELECTX() below for details on usage.

SelectX()

Applies to: templates

This sequence of tags, going from SELECT1() to SELECT99(), if necessary, provides a means of allowing the user to select from a variety of traits, for adding to their character, in the manner of character templates.

These tags should be used in numerical order, starting from SELECT1(). GCA will stop processing more SELECTX() tags as soon as it fails to find the next number in the sequence.

When used inside of a TRIGGERS() tag, SELECT() may be (and should be) used without a numeric index, because the order is determined by the order found in TRIGGERS() rather than by the index given.

SELECTX() basically has GCA create a specialized Item window, which allows the user to select from a custom list of system traits. The traits selected are added to the character. Some number or point value of items necessary may be specified, similar to #CHOICELIST(). Because you're working with actual items in the window, it's possible to have them built using modifiers, and to adjust their levels and such.

```
SELECTX( TEXT(DIALOG TEXT),
    MULTITYPE(YES),
    CONDITIONAL(EXPRESSION),
    ITEMSWANTED( EXACTLY | UPTO | ATLEAST ] NUMBER [, EXACTLY | UPTO | ATLEAST ] NUMBER ] ),
    POINTSWANTED( [ EXACTLY | UPTO | ATLEAST ] COST),
    RECOMMENDED(RECOMMENDEDTRAITLIST)
    TAGWITH(TAG VALUES),
    LIST(TRAIT [, #NEWITEM(DEFINITION) ] [ #CODES(CODES) ] [ #TAGS(REPLACEMENTS) ] [ #NOTE(NOTES) ] )_
)
```

Since this is a tag list, most of the tags are optional.

SELECTX

The current SELECTX item sequence.

TEXT(DIALOG TEXT)

The explanatory text to be displayed to the user in the Item dialog they will be presented with.

MULTITYPE(YES)

If the LIST() includes more than one type of trait type (such as skills mixed with advantages), then you should usually include MULTITYPE(YES), so that GCA will structure the display to be more neutral in how it displays the traits.

You may use of MIXEDTYPE(YES) as a synonym for MULTITYPE(YES).

CONDITIONAL(EXPRESSION)

CONDITIONAL() allows you to provide an expression to evaluate, and that expression must evaluate to True (a non-zero result) in order for the SELECT() to be processed.

```
        select(_
                text("Select with Conditional. You'll never see this one."),
                conditional(@sametext("alpha", "bravo")),
```

In this example, @sametext("alpha", "bravo") evaluates to 0, so the Select dialog would not be shown.

CONDITIONAL() is math enabled, so it supports the full Solver functionality to determine if the conditional applies.

ITEMSWANTED(EXACTLY | UPTO | ATLEAST] NUMBER [, EXACTLY | UPTO | ATLEAST] NUMBER])

ITEMSWANTED() allows you to specify what number of items the user is supposed to select. You may specify just the NUMBER by itself, or you may use an optional qualifier as shown. If you use the UPTO or ATLEAST qualifier, you may specify a second number of items wanted, with a second qualifier, to limit the range introduced. Separate the first block from the second with a comma.

For example, if you want the user to pick at least one option, but they may pick as many as five, you'd use itemswanted(atleast 1, upto 5).

If no qualifier is specified, EXACTLY is assumed.

This tag is math-enabled.

POINTSWANTED([EXACTLY | UPTO | ATLEAST] COST)

POINTSWANTED() allows you to specify that you want some number of points to be required. You may use the optional qualifiers to provide more leeway in selection. If no qualifier is specified, EXACTLY is assumed.

This tag is math-enabled.

RECOMMENDED(RECOMMENDEDTRAITLIST)

RECOMMENDED() allows you to specify one or more traits that are recommended to the user. These traits will be flagged with a star icon in the Available list.

RECOMMENDEDTRAITLIST should be a list of trait names in the usual fashion, with prefix tags and enclosed in quotes or braces if necessary. Of course, these should also be traits that are expected to appear in the LIST() that the dialog will display to the user.

You can use RECOMMEND() if you prefer.

TAGWITH(TAG VALUES)

TAGWITH() allows you to tag any traits added/modified by the SELECTX()with one or more tags and values after OK is clicked on the Select dialog.

TAG VALUES should be a list of tags with values. For example

```
tagwith(Barbarian(yes), PrimaryWeapon(Barbarian))
```

will tag all traits taken with two custom tags, Barbarian() and PrimaryWeapon(), with values of yes and Barbarian, respectively.

LIST(TRAIT [, #NEWITEM(DEFINITION)] [#CODES(CODES)])

LIST() is the list of traits to be presented to the user for selection. These traits should be specified by name and prefix tag, as appropriate. As with all such lists, it should be comma separated, and items should be enclosed in braces or quotes as required.

TRAIT	The standard trait reference, specified by name and prefix tag.
#NEWITEM(DEFINITION)	This special directive allows you to define a brand new item, from scratch, as if it existed in the data files. (It does not exist in the data file! Defining it here does not create an item in the data file and does not allow for a reference to it elsewhere.) The DEFINITION should be in the same format as used in the appropriate section of the data file, with the exception that it must begin with the appropriate prefix tag, so that GCA will know what type of trait it is. (GCA will reject #NEWITEM() commands that don't include the proper prefix tags.) As one of the tags defined for the DEFINITION you may include the EXISTING(EXISTINGTRAIT) tag to specify that this new item already exists *on the character*. The EXISTINGTRAIT specified must include the full name and prefix tag for the existing item, so that GCA can find it correctly; if it's not found, the item will not be presented to the user.

If the DEFINITION also includes ADDMODS() and/or REMOVEMODS() in the tag list, GCA will apply them to the found ExistingTrait immediately, before presenting the trait in the Select dialog.

Note: GCA no longer requires you to use EXISTING() in order to access traits already on the character. Traits that are already taken by the character will be flagged for the user in the Select dialog's Available list, and if added to the Selected list, they'll reflect the trait as it is on the character, including any already spent points, and adjusted by any #CODES specified.

#CODES(CODES) Allows you to include special codes that control how the trait is handled. You may include the #CODES directive with a TRAIT or a #NEWITEM(DEFINITION) by simply tacking it on at the end of the reference, outside of any quotes or braces, and without a comma, so that GCA knows it applies to the item it's next to, but that it's not a new item.

You may use two possible code items: UPTO VALUE and DOWNTO VALUE. UPTO specifies an upper limit for the trait, and DOWNTO specifies a lower limit. If VALUE includes PTS, then the limit value is considered to be a number of points. (PTS is only applicable only to skills and spells, it won't work for any other trait types.) If using both UPTO and DOWNTO, separate with a comma. VALUE is math enabled.

There are a few special case variables you may use as part of the VALUE: %POINTS which references the associated trait's current points, %LEVEL which references the associated trait's current level, and %SCORE which references an attribute's score (only valid for attributes).

#TAGS(REPLACEMENTS) Allows you to replace the contents of any tags on the library item being referenced, before the item is sent to the Select dialog. List all the tags desired inside the #TAGS() parentheses, bearing in mind the usual nesting requirements. Include this special directive with an item in the list by simply tacking it on at the end of the reference, outside of any quotes or braces, and without a comma, so that GCA knows it applies to the item it's next to, but that it's not a new item.

#TAGS() allows you to customize items without having to use #NEWITEM, if the item is one that can be easily based on an existing library trait. Replace NAME(), NAMEEXT(), INITMODS(), whatever you need—just remember that the tags must be Library tags because the items shown in the Select dialog are Library items until added to the character. Also bear in mind that you can not add new attack modes using NEWMODE(), because NEWMODE() tags are processed when the book data is loaded, long before this point.

For example, if you have a SELECTX() with a LIST() like this:

```
list(_
SK:Performance #tags(nameext(Singing), page(TESTING))#codes(upto 16, downto 10),_
SK:Photography #tags(name(Holography), page(TESTING))#codes(upto 16, downto 10)_
)_
```

the user will be presented with a list of options that includes Performance (Singing) and Holography/TL (because Photography is a /TL skill). Aside from the name change and the new TESTING page number, everything else for those two skills will be identical to Performance and Photography, respectively.

#NOTE(NOTES) Allows you to specify a special note to include on the selection list user interface. Include this special directive with an item in the list by simply tacking it on at the end of the reference, outside of any quotes or braces, and without a comma, so that GCA knows it applies to the item it's next to, but that it's not a new item.

This special note will be displayed directly under the trait name on the list, and that list entry will be taller to accommodate it.

Now that you know the parts, here are some examples.

Example 1:

```
select4(_
        text("Select the disadvantages you want for your character from the list below. _
        You need to select -35 points worth."),
        pointswanted(-35),
        itemswanted(atleast 1),
        list(_
                DI:Alcoholism,
                DI:Flashbacks #codes(UPTO 1),
                DI:Honesty,
                DI:Overconfidence,
                #newitem(_
                        DI:Sense of Duty (Comrades), -5,
                        page(B153),
                        cat(Mundane, Mental)_
                        ),
                DI:Trademark #codes(UPTO 1)_
        )_
)
```

This example is a relatively simple SELECTX()that gets some disadvantages for a template from the user. Note that it uses a #NEWITEM() to define a Sense of Duty (Comrades) disadvantage. Also note how the #CODES() directives are used to limit the traits to just one level, and that they're listed along with the trait references, not separated from them by commas.

Example 2:

```
select1(_
    text("Please select two knightly weapon skills to train."),
    tagwith(knightly(yes)),
    pointswanted(exactly 8),
    itemswanted(exactly 2),
    list(_
        SK:Broadsword #codes(upto 4pts, downto 4pts),
        SK:Axe/Mace #codes(upto 4pts, downto 4pts),
        SK:Bow #codes(upto 4pts, downto 4pts)_
    )_
)
```

This example includes the TAGWITH() tag, which allows the template to tag the selected weapon skills with a knightly(yes) tag when the user is finished. That tag might then be used later in a #BUILDSELECTLIST to target those traits, like this:

```
select3(_
text("Please select an appropriate number or value of these traits."),
pointswanted(exactly 4),
itemswanted(exactly 1),
list(_
    #BuildSelectList(Skills where knightly is "yes",
            template(_
                    #newitem(SK:Increase %ListItem% by 4 points, existing(SK:%ListItem%)) _
                    #codes(upto %points+4pts, downto %points+4pts)_
                    )_
    )_
)_
)
```

Result Variables

SELECTX() dialogs will now return the number of items selected and the points spent, and the number of items unselected and points unspent when it's possible to determine that, when the user finishes the dialog. This allows for referencing those values in other SELECTX() dialogs down the line.

Note that if ATLEAST is specified for points or count, the unused or unspent values will always be zero.

To use these values, you will need to add a NAME() tag into the tag list for the Select(), and specify a name. If you don't specify the NAME(), the default name is just the number of the SELECTX() in the processing order (if you used SELECTX() outside of TRIGGERS(), X will be the number, otherwise it's counted from 1 with the first SELECT() dialog). You can then reference the desired value in the POINTSWANTED() or ITEMSWANTED()/NUMBERWANTED() tags, just as you would with a #CHOICELIST substitution, using the name of the SELECTX() and the desired value within % signs.

The variables are: %XUNSPENTPOINTS%, %XSPENTPOINTS%, %XUNUSEDSLOTS%, %XUSEDSLOTS%, %XNUMBERUNUSED%, and %XNUMBERUSED%.

The X in each is replaced by the SELECTX() NAME() value.

%XUNSPENTPOINTS% and %XSPENTPOINTS% return unspent and spent points, respectively, for the specified SELECTX().

%XUNUSEDSLOTS% and %XNUMBERUNUSED% will both return the allowed number of picks that went unused, and %XUSEDSLOTS% and %XNUMBERUSED% will both return the number of picks actually used.

So, a SELECTX() with name(Alpha) would use %alphaunspentpoints% in a subsequent SELECTX() POINTSWANTED() tag to reference those unused points, maybe something like pointswanted(30 + %alphaunspentpoints%) to allow for those unused points to be spent here.

The Difference Between #Choice/#ChoiceList and SelectX()

Many people are confused by the difference between #CHOICE/#CHOICELIST and the SELECTX() tags. The important difference to remember is that the #CHOICE/#CHOICELIST directives deal with *text*, while SELECTX() deals with *traits*. The results placed into the result variables by #CHOICE/#CHOICELIST are just text values, exactly as specified in the directive; the result of the SELECTX() selections are traits added to the character.

SetLoadout()

Applies to: templates

SETLOADOUT() can be used to change the character's active loadout to the loadout specified in the tag.

This tag can be used inside TRIGGERS().

Sets()

Applies to: templates

This tag allows a template to set traits to a given value.

For example:

```
sets(ST:IQ=15, ST:Hit Points=12)
```

would set IQ to 15 and Hit Points to 12. This is just as if the user had done it, so point costs and such would apply as appropriate.

ShortLevelNames()

Applies to: modifiers

Like LEVELNAMES(), but shorter. Intended for use in places where shorter versions are helpful, and where the full-length name isn’t needed.

ShortName()

Applies to: modifiers

SHORTNAME() is intended to include a short, to-the-point name for the modifier, taking up as little space as possible. This is better for display in attribute blocks (and in item lists generally) than the longer name that's more useful for making selections from dialogs.

Shots()

Applies to: traits with damage modes

This tag specifies the Shots value of an attack or other effect. This should be a simple numeric value, possibly with a simple text suffix.

This tag is mode-specific.

SkillUsed()

Applies to: traits

This tag contains a list of traits, and any adjustments necessary if a particular trait is used, designating the skill used for the attack or feature. For example,

> skillused(DX, Brawling, Karate)

specifies that any of the three traits listed may be used, and each is used at full value. And

> skillused(Brawling-2, Kicking (Brawling))

specifies that the character can use Brawling at -2 or Kicking (Brawling) at full value.

Many data files do not use prefix tags in this tag, but their use is strongly encouraged, nevertheless. If the trait name includes a comma, or a + or - character, be sure to enclose it in quotes.

You can specify that GCA use the worst of the listed skills, instead of the best, using the #WORST special directive somewhere in the list of traits (I suggest as the first thing). Do not separate the directive from the list with a comma, as GCA will identify it and remove it before parsing the list. This flag must be in the SKILLUSED() list for a mode in order to apply, otherwise GCA will continue to select the best possible skill for use.

> skillused(#Worst DX, Brawling, Karate)

In this example, GCA will take the worst of the three possible choices. Note that the #WORST directive is just inserted into the first item of the list without a comma.

This tag is mode-specific.

Stat()

Applies to: skills and spells

This tag is used if the skill is a based on a different attribute than the normal attribute for its type. Specify the attribute to be used as the value of the tag.

STCap()

Applies to: traits

The way GCA supports the maximum ST cap for muscle-powered weapons has changed. GCA will now apply a ST cap to all muscle-powered weapons, regardless of type.

Because it's now easier for the cap to be applied to weapons that perhaps aren't actually meant to be capped (either by RAW or in your campaign), GCA now supports the STCAP() tag to turn off capping, or to change the multiplier value if desired.

Use STCAP(NO) to turn off the max ST cap:

> stcap(no)

or use STCAP(X), replacing X with a value, to replace the 3*ST cap with an X*ST cap:

> stcap(2)

Math-enabled. Mode-enabled.

Step()

Applies to: attributes

This tag specifies the step value for the attribute. The step value is the amount by which the attribute's score changes for each level raised or lowered. For example, step(1) is the value for attributes such as ST and DX, where adjusting the level by one likewise changes the score by one. On the other hand, Basic Speed uses step(0.25), instead, because every five points changes the level a quarter.

This tag is math enabled.

SubsFor()

Applies to: traits

This tag supports the new Substitutions system, which allows one trait to substitute for another when GCA is tracking down references. This system applies to character traits and does not do substitutions on a library basis.

> SUBSFOR(TRAITNAME [, TRAITNAME [, ...]])

TRAITNAME must be a fully qualified trait name, including a prefix specifying the trait type. Multiple traits may be defined by separating them with commas. Traits should be enclosed in quotes or braces as required.

See the Substitutions *section in* Special Notes *for more information.*

Symbol()

Applies to: attributes

This tag provides attributes with an alternative name for use in calculations. Usually shorter and easier to reference, the calculation symbol is only supported in math expressions. The SYMBOL() value should be simple text, and should never use restricted characters.

Taboo()

Applies to: traits

This tag allows you to specify taboo items for the trait. This tag supports the same basic features as the NEEDS() tag. Remember that these things are *not* wanted, so any found will result in a warning marker in GCA.

TargetListIncludes()

Applies to: traits

This tag supports the new “user targetable” bonuses feature, to specify the types of target traits that are applicable to a trait that supports such bonuses.

File authors can suggest and limit the available targets presented to the user by using the TARGETLISTINCLUDES() tag on the item, which includes one or more criteria for adding traits to the list, using a structure similar to that in the #BUILDSELECTLIST directive. It's structured like this:

```
TARGETLISTINCLUDES( TRAITTYPE [ WHERE TAG [ IS | ISNOT | INCLUDES | EXCLUDES | LISTINCLUDES |
LISTEXCLUDES ] TAGVALUE ][ , ... ] )
```

which allows for specifying just the trait’s type in TRAITTYPE if that's the only limit required. If more restriction is needed, see the docs for #BUILDSELECTLIST to see how the WHERE block works. Note: GCA will never include locked or hidden traits as valid targets. In addition, TRAITTYPE may be ANY or ALL to specify selecting from all traits, not just one type.

As an example, instead of requiring users to add a Weapon Master Damage Bonus modifier to every weapon that gets the bonus, the bonus can be included with the Weapon Master advantage, and can use the new %CHOSENTARGET% target instead. The user can then select or modify the weapons that apply whenever they need to do so in the Edit window.

Here's a modified version of Weapon Master:

```
Weapon Master (Targets), 20/25/30/35/40/45, levelnames(one specific weapon, two weapons normally used
together, a small class of weapons, a medium class of weapons, a large class of weapons, all muscle
powered weapons), page(B99), upto(6), cat(Mundane, Physical),
	x(_
		#InputToTagReplace("Please specify the weapon, weapons, or class of weapons you have
Mastery of:", nameext, , "Weapon Master")_
		),
	gives(_
		=+@if(_
			$modetag(charskillscore) = ST:DX+1 _
			THEN @textindexedvalue($modetag(dmg), ("thr", char::basethdice), ("sw",
char::baseswdice), ELSE $solver(me::dmg)) _
			ELSE @if(_
				$modetag(charskillscore) > ST:DX+1 _
				THEN @textindexedvalue($modetag(dmg), ("thr", 2 * char::basethdice),
("sw", 2 * char::baseswdice), ELSE 2*$solver(me::dmg)) _
				ELSE 0 _
```

```
                                        )_
                              ) to %chosentarget%::damage$ listas Weapon Master Damage Bonus _
                       ),
         targetlistincludes(Equipment where charreach isnot "", Equipment where charrangemax isnot "")
```

TechLvl()

Applies to: equipment

This tag specifies the first tech level at which the equipment item becomes available. This should be a simple numeric value, sometimes with a ^ marker. Sometimes a plain text value such as ^ or var. is used, instead.

Tier()

Applies to: modifiers

This tag specifies the calculation tier for the modifier. The default tier is 0, and tiers may range from -2 to +2.

When GCA calculates modifiers, it calculates them in tiers so that special cases may be handled before or after the modifiers that are being handled normally. GCA calculates each tier in succession, from -2 to +2, using the final value from the previous tier as the base starting value for the next tier.

TL()

Applies to: traits

This is the tech level for the trait. This may be a simple numeric value, or a simple range, such as tl(8) or tl(3-5).

When added to the character, GCA will assign a value to this tag for the character trait, and the value assigned will be either the given TL() value, or if a range is available, a value equal to the character's TL within the range, or the end of the range closest to the character's TL. For example, if the character's TL is 8, and the trait has tl(3-5), the character trait will get the TL() tag of tl(5).

Equipment: TL() for equipment items should not be a range, but a single TL value. If a TL() exists on the item when added to the character, that is used as-is for the TL of the item. If not, then a TL() tag will be created similar to the way explained above, but it uses the TECHLVL() tag as the floor for determining the TL assigned.

Triggers()

Applies to: templates

GCA now supports a more flexible means of handling a variety of template features.

The TRIGGERS() tag is a wrapper/processor tag that contains all of the various "command" tags, or tags that are "triggered" by an item being added to a character. This new tag allows for all of those trigger tags to be included and processed in the order desired.

In addition, you may include any number of each tag that you may wish. That means you could include an ADDS() then a REMOVES() and then another ADDS(). You may also include SELECT() tags, which no longer need to be numbered sequentially; they'll be processed in the order they're included within the TRIGGERS() tag, as they're encountered. (If you do number them, GCA will ignore the numbers and still process them in the order listed.)

The syntax for each of the trigger tags within TRIGGERS() remains exactly the same.

The existing system for the trigger tags still remains, and still works in the exact same fashion it always has, with the exact same processing order, and the same limitation of one tag of each type. (And for that system, you must still use the sequentially numbered SELECTX() tags, as well.) The new TRIGGERS() tag contents will be processed first, **before** any of the old system tags are processed.

Here are all the trigger tags currently recognized by GCA:

ADDMODS(), ADDS(), BODYTYPE(), CHARHEIGHT(), CHARWEIGHT(), CHILDOF(), CREATES(), LOADOUT(), MERGETAGS(), PARENTOF(), RACE(), REMOVEMODS(), REMOVES(), REMOVESBYTAG(), REPLACETAGS(), SELECT(), SELECTX(), SETS().

Processing of the various tags within TRIGGERS() supports Text Function Solver processing of each tag's contents before it is evaluated. This means, for example, that you could use an $IF() function to dynamically allow for different possible versions of the same tag, based on other values.

In addition, processing of the various sub-tags also supports #BUILDSELECTLIST processing of each tag's contents. In most cases, #BUILDSELECTLIST isn't necessary, or even makes no sense at all. However, it does allow for one way of dynamically changing the intended content of trigger tags based on previous activity in the TRIGGERS() tag sequence.

Type()

Applies to: skills and spells

This tag specifies the type of the skill or spell, as available from the SkillTypes section. For skills, this tag is usually not used, as the type is part of the normal definition that doesn't require the TYPE() tag, but for spells, this tag must be used to specify a type that isn't the default.

Units()

Applies to: traits

Another one that's not technically new, UNITS() has been used primarily to support labeling the trait's primary value with an applicable unit of measurement. This tag specifies the units for the primary value of the trait, not necessarily for the various sub-features. For example, equipment items always use UNITS() to specify the weight units of the item, and if it's not found, pounds (lb) are assumed.

Always use the abbreviation, such as UNITS(m) or UNITS(kg), as that's the intended format. Some new features inside GCA will check this tag to find a conversion method, and if it's not found in the expected format it may result in unintended behavior.

You can specify both the standard and metric units by separating them with a | character, such as UNITS(lb|kg). In this case, GCA will always use the first unit if the character is standard and the second if the character is metric. If specifying both units, the trait must be sure to handle standard or metric conversion itself, so that when the character is metric, the value is also metric.

GCA currently does automatic conversion to/from metric values only for equipment weights and weight capacities. Currently supported conversions use the ‘good enough’ rounded values from Basic Set.

The currently supported units in the converters are: **in** (inch), **cm** (centimeter), **ft** (foot), **y** (yard), **yd** (yard), **m** (meter), **mi** (mile), **km** (kilometer), **lb** (pound), **lbs** (pound), **kg** (kilogram), **oz** (ounce), **g** (gram), **gal** (gallon), **liter** (liter), **l** (liter), and **qt** (quart). Not all are used at this time.

When GCA processes items, it will create the CHARUNITS() tag to specify what units currently apply.

Up()

Applies to: attributes

This tag specifies the cost for each step incremented above the base value of the attribute. This may include multiple costs, separated by slashes, if the cost isn’t the same for each level. GCA will use the difference between the last two costs as the cost per level for any additional levels beyond the given progression.

All values specified in the UP() tag must be simple numeric values.

For example, up(5) specifies a cost of 5 points per level, while up(5/10/20/40) specifies costs increasing from 5 points for the first increment, to 20 points per increment after the third.

UpFormula()

Applies to: attributes

Allows you to specify a formula to use for calculating the cost of the attribute when raising the attribute from the base value. Math enabled.

UpTo()

Applies to: traits

This tag is math enabled.

Languages, Cultures, Advantages, Perks, Features, Disadvantages, Quirks, and Templates; Modifiers

```
UPTO(VALUE [ LIMITINGTOTAL ] )
```

This tag specifies the maximum number of levels allowed for the trait.

If LIMITINGTOTAL is specified as well, then GCA will also try to limit the levels taken to include any bonuses that have been applied. Normally, bonuses are allowed to raise the level beyond the UPTO() value.

Skills and Spells

UPTO(VALUE[PTS])

This tag specifies the maximum level allowed for the trait.

If PTS is specified, then the UPTO() value is actually the maximum number of points that may be spent in the skill.

Equipment

UPTO(VALUE)

This tag specifies the maximum item count allowed for the equipment item.

UserNotes()

Applies to: traits

USERNOTES () is the tag to use when you want to include longer text notes with a trait, and it's not information that you want to include in the DESCRIPTION().

Note that NOTES() was originally defined and meant for use with the numeric reference indexes on weapon tables and the like, so that information was meant to fit in a narrow column of one-half inch or so in width. Many data file authors didn't realize that, however, and used NOTES() for general trait notes. Please use USERNOTES() for general trait notes, and ITEMNOTES() if you want to add longer notes for particular modes.

For displaying very short info, such as the energy cost range of a spell, using NOTES() is a fine alternative that will then fit nicely in the Notes column of an attack table or a grimoire.

RTF: GCA supports using RTF text inside this tag, instead of the plain text used in most other places. GCA provides a rudimentary RTF editor to allow you to change text colors and fonts, or to bold or italicize words in the text, but if you want to get fancier you may want to write things in WordPad or another RTF editor and then paste the RTF into this tag.

While GCA internally allows for carriage returns (CR) and line feeds (LF) within this text, the GDF format does **not**. Within the book data, this text must still fall back to a single line after line continuation characters. This is important because RTF frequently includes line feed (LF) characters within the RTF text, usually after paragraph or newline codes.

You can manually replace CR+LF, CR, or LF characters with special codes that will be replaced by GCA with the corresponding characters upon loading the data. Use <CRLF> for CR+LF, <CR> for CR, and <LF> for LF. Be sure that you actually remove the offending control characters after you insert the codes, otherwise your trait will not load correctly.

Uses()

Applies to: traits

USES() was originally only used by Phoenix, but is now fully supported by GCA. This allows for specifying how many uses (doses, shots, bullets, etc.) are available to a trait, and to create check boxes for them. The special keyword COUNT is supported to reference the quantity or level of the item or trait with the USES() tag.

The Uses subsystem is mode specific, so having one will create at least one mode even if no other mode data is used.

You may specify more complex math (if desired) by forcing the use of the Solver. If desired, just begin the text of the tag value with an = sign, such as =me::score*2.

Note: Don't mix COUNT and = because the simple replacement method of COUNT with a value may conflict with the Solver's ability to find trait references if COUNT was incorrectly replaced with a value inside of such a reference.

Uses_sections()

Applies to: traits

USES_SECTIONS() allows for specifying how many groups of the USES() check boxes to create, which are shown in different sections in the display area. This was called USESCOUNT() by Phoenix, but I wanted to have all the sub-tags use underscores and thought this would more clearly show how I see it working. I have set up aliases in GCA to allow using the USESCOUNT() tag to access the USES_SECTIONS() tag data. GCA should handle that transparently, but will save it out to the save files as the USES_SECTIONS() tag.. The special variable COUNT is supported as per USES().

You may specify more complex math (if desired) by forcing the use of the Solver. If desired, just begin the text of the tag value with an = sign, such as =me::score*2.

Note: Don't mix COUNT and = because the simple replacement method of COUNT with a value may conflict with the Solver's ability to find trait references if COUNT was incorrectly replaced with a value inside of such a reference.

Uses_settings()

Applies to: traits

USES_SETTINGS() contains a USES() specific set of tags that specifies display and operational settings for the USES subsystem and controls. These tags should be formatted exactly like any other tag list, separated by commas. For example:

```
usersettings( selectioncolor(orange), trackusesbysection(False), uselargerboxes(True) ).
```

This example specifies that the USES selections should be orange, they should not be tracked per section, and that you'd like to use larger checkboxes.

There are a small number of built-in 'styles' of checkboxes, which can be specified for use in the ALTBOX() tag within the USES_SETTINGS() tag. The default style name is black, and there are also red, orange, yellow, gold, magenta, green, and blue styles. Each of these styles specifies the color of the box, the interior of the box, the tick used to check the box, and a couple other bits. Right now, I've defined all that with these default built-ins, but some day it should be customizable somewhere.

The ALTBOX() tag allows you to specify the box style, and with which box number that style begins. Styles apply starting at the specified box and continuing on from there until either the boxes end or another is defined. For example,

```
altbox(green, 10, red, 20)
```

would start with the default boxes until box 10, where they change to green, and stay green until box 20, where they become red.

Note that some of the style colors can be hard to differentiate, especially on a white background.

Uses_used()

Applies to: traits

USES_USED() tracks how many uses have been used of the USES() available. If tracking uses by section, then this may be a list of numbers separated by + signs, such as 4+0+1 to show that there are three sections, and there have been 4 uses used in the first section and 1 uses used in the last section. If not tracking uses by section, then even if you have different sections specified, all uses are consumed as if from one big pile. Tracking uses by section is currently the default behavior.

Vars()

Applies to: traits

GCA now supports defining variables at the trait level. Variables are defined using the VARS() tag, like so:

```
VARS( NAME1 = VALUE1 [, NAME2 = VALUE2 ] [, ... ] )
```

This allows for greatly reducing the complexity of certain types of formulas, such as the new DISPLAYNAMEFORMULA().

These are simple substitution variables; if you were to use

```
vars(%name% = me::name)
```

then the %name% variable stores the text me::name, **not** the actual name of the trait.

For example, to create a DisplayName that uses the same name that GCA would generate, but adds the additional text fragment Bonus Text **within** the parenthetical information (if any), the straight formula might look like this:

```
displaynameformula($if(@endswith($val(me::basedisplayname), %closeparen) then
$insertinto($val(me::basedisplayname), "; Bonus Text", @len($val(me::basedisplayname))-1) else "(Bonus
Text)" ))
```

Notice that we have to keep repeating the $val(me::basedisplayname) bit over and over again. (And $VAL() is necessary, because the Text Function Solver won't replace any value references unless explicitly told to do so with the $VAL() or $TEXTVALUE() functions). Just imagine the additional complexity if we wanted to include a trait value or tag reference instead of the simple constant Bonus Text.

So, we can simplify with a variable:

```
vars(%name% = $val(me::basedisplayname))
```

and the new DISPLAYNAMEFORMULA() that makes use of it:

> displaynameformula($if(@endswith(%name%, %closeparen) then $insertinto(%name%, "; Bonus Text", @len(%name%)-1) else "%name% (Bonus Text)"))

In this case, not drastically shorter as text goes, but much more readable.

Because GCA will replace the variable name indiscriminately within the target area as the first step of evaluating an expression, you should ensure that your variable names are unlikely to conflict with other types of text. You must begin each variable with a percent sign %, and I recommend using a percent sign % at the end of each variable name also, to ensure no accidents are likely to occur. You should also avoid variable names of %LEVEL and %COUNT since those are special cases built into GCA.

VTTNotes()

Applies to: traits

Provides a place to enter notes or formulas specific to the virtual tabletop program that you use.

For general trait VTT notes, use VTTNOTES(). For mode-specific notes, use VTTMODENOTES().

VTTModeNotes()

Applies to: traits

Provides a place to enter notes or formulas specific to the virtual tabletop program that you use, for the particular mode in question.

For general trait VTT notes, use VTTNOTES(). For mode-specific notes, use VTTMODENOTES().

Mode-enabled.

WeightCapacity()

Applies to: traits

The weight capacity of the item.

When GCA processes the item, it will create these tags: WEIGHTCAPACITYLEVEL(), which shows the percentage of the current capacity used; OVERWEIGHTCAPACITY(), which is just YES if the item currently exceeds its capacity; OVERWEIGHTCAPACITYBY(), which contains the amount by which capacity is exceeded.

WeightCapacityUnits()

Applies to: traits

The units that apply to the given WEIGHTCAPACITY(). This works the same way as the UNITS() tag, but is specific to the WEIGHTCAPACITY() of the item.

When GCA processes the item, it will create the CHARWEIGHTCAPACITYUNITS() tag to show the currently applicable units.

WeightFormula()

Applies to: equipment

This tag allows for specifying an expression that should be evaluated to determine the weight of an item. If a WEIGHTFORMULA() tag is included, the weight will always be calculated based on the given formula. This tag overrides much of the normal behavior of calculating item weight, overriding anything calculated using modifiers and child items. For this reason, it may be better to use BASEWEIGHTFORMULA() for the base weight, unless you actually want a full override for some reason.

Where()

Applies to: equipment

The WHERE() tag is in support of 'Where It's Kept' on the character, since LOCATION() is used for armor coverage. This is for those who like to detail out where each item is placed on their character.

Calculated Tags

Tags that GCA uses to store calculated values. You can use these by reference inside other tags, and sometimes use them as bonus targets, but setting these tags directly will not work because GCA replaces the value constantly with newly calculated values or deletes the tags when not applicable.

CharFlexible()

Calculated. Applies to: equipment with DR

The calculated character-specific version of FLEXIBLE().

If the armor is determined to **not** be flexible CHARFLEXIBLE() will not exist and will be deleted if it did exist. GCA will set the value to yes if the armor is flexible. In parallel, GCA will add or remove the * marker to the end of the DR value to match the flexible status (with * meaning flexible).

As with all other calculated character-specific calculated features, exporters and sheets will want to use CHARFLEXIBLE() if they need it for something, but any flexible armors will still be marked with the * for user reference. If those values are also used, you may need to remove the * from the end.

ChildrenCosts()

Calculated. Applies to: equipment

The total cost of all the child items, after adjustment by any bonuses.

Targetable by bonuses.

ChildrenWeights()

Calculated. Applies to: equipment

The total weight of all the child items, after adjustment by any bonuses.

Targetable by bonuses.

PreChildrenCost()

Calculated. Applies to: equipment

The cost of the item after FORMULA()/COSTFORMULA() was applied. If no formula was used, then it will be the same as PREFORMULACOST().

PreChildrenWeight()

Calculated. Applies to: equipment

The cost of the item after WEIGHTFORMULA() was applied. If no formula was used, then it will be the same as PREFORMULAWEIGHT().

PreCountCost()

Calculated. Applies to: equipment

The cost of the item after modifiers are calculated but before being multiplied by COUNT().

PreCountWeight()

Calculated. Applies to: equipment

The weight of the item after modifiers are calculated but before being multiplied by COUNT().

PreFormulaCost()

Calculated. Applies to: equipment

The cost of the item after COUNT() was applied to PRECOUNTCOST(), before FORMULA()/COSTFORMULA() possibly throws that all away with an overriding formula.

PreFormulaWeight()

Calculated. Applies to: equipment

The weight of the item after COUNT() was applied to PRECOUNTWEIGHT(), before WEIGHTFORMULA() possibly throws that all away with an overriding formula.

PreModsCost()

Calculated. Applies to: equipment

The cost of the item after owned components are included, but before modifiers are calculated.

PostFormulaCost()

Calculated. Applies to: equipment

The cost of the item after FORMULA()/COSTFORMULA() was applied. If no formula was used, then it will be the same as PREFORMULACOST().

PostFormulaWeight()

Calculated. Applies to: equipment

The weight of the item after WEIGHTFORMULA() was applied. If no formula was used, then it will be the same as PREFORMULAWEIGHT().

MATH

Many tags in GCA are *math-enabled*, meaning that GCA will evaluate a given expression to find the desired value. Because of this, GCA must be able to see and recognize math characters for what they are. There are a number of characters that are recognized as math characters in such tags, and you should take special care. These characters are all recognized as math characters:

( ) + - / * = > < & | ^ \

If any of these characters appears within the name of an item that is being used in a math section, that item name must be enclosed within double quotes, including any prefix tag, if applicable. For example, if a weapon skill was to default from the skill Axe/Mace at -4, then the default( ) tag for that item should look like this:

```
default("SK:Axe/Mace"-4)
```

Notice that the prefix tag of SK: is included within the quotes along with the name of the skill, but the rest of the math expression appears outside of the quotes.

Math Functions

There are a number of math functions supported where math can be used. These functions are described below.

@BaseSWDice

This function takes a value and returns the base Swing damage dice that would result if the value was a ST score.

```
@BASESWDICE(VALUE)
```

Where:

VALUE	is an expression that evaluates to a number.

@BaseTHDice

This function takes a value and returns the base Thrust damage dice that would result if the value was a ST score.

```
@BASETHDICE(VALUE)
```

Where:

VALUE	is an expression that evaluates to a number.

@BonusAdds

```
@BONUSADDS(TAGNAME)
```

Returns the total of all add and subtract bonuses/penalties targeted to the given tag of the calculating trait, or 0 if there are none.

@BonusMults

> @BONUSMULTS(TAGNAME)

Returns the multiplicative total of all multiplier functions targeted to the given tag of the calculating trait, or 1 if there are none.

@Ceiling, @Ceil

> @CEILING(VALUE)
> @CEIL(VALUE)

Returns the smallest integer value that is greater than or equal to the given VALUE.

@EndsWith

> @ENDSWITH(DOESTHIS, ENDWITHTHIS)

Returns 1 if DOESTHIS ends with ENDWITHTHIS, 0 if not. Case is ignored.

If you want to check for parens ( ), braces { }, or quotes ", you can instead use these variables as needed: %CLOSEPAREN, %OPENPAREN, %CLOSEBRACE, %OPENBRACE, %QUOTES.

Note that the %CLOSEPAREN, %OPENPAREN, %CLOSEBRACE, and %OPENBRACE special variables should not be necessary if the corresponding characters are enclosed within quotes in the function, such as

> @endswith(")")

The %QUOTES variable is still required, as there is no way to enclose the double quote character safely, but it's also the least likely to be needed for anything.

@Fac

This function takes a value and returns the factorial value of that number.

> @FAC(VALUE)

Where:

VALUE	is an expression that evaluates to an integer.

You may use @FACTORIAL as a synonym for @FAC.

@Fix

This function returns the integer portion of a number.

> @FIX(VALUE)

Where:

VALUE	is an expression that evaluates to a number.

The difference between @FIX and @INT is in negative numbers: @INT returns the first negative integer less than or equal to VALUE, while @FIX returns the first negative integer greater than or equal to VALUE.

@Floor

@FLOOR(VALUE)

Returns the largest integer value that is less than or equal to the given VALUE.

@HasMod

This function is used to determine if the containing trait has a certain modifier applied to it.

@HASMOD(MODNAME)

Where:

MODNAME	is the name of a modifier.

This function returns the level of the modifier specified, if the modifier is applied to the containing trait. Otherwise it returns 0.

This function will look for modifiers that have name extensions using either the old NAME (NAME EXT) format, or the newer NAME, NAME EXT format that modifiers display in the UI now.

@HasModIncludesText

Searches the item's modifiers (name and name extension only, and each is checked separately) for all those that include the text specified and returns the highest level among those that include it.

@HASMODINCLUDESTEXT(TEXT)

Returns 0 if none found.

@HasModIncludesTextMin

Searches the item's modifiers (name and name extension only, and each is checked separately) for all those that include the text specified and returns the lowest level among those that include it.

@HASMODINCLUDESTEXTMIN(TEXT)

Returns 0 if none found.

@HasModWithTag

Searches the item's modifiers for those that match a given value for a given tag and returns the highest level from those found.

@HASMODWITHTAG(TAGNAME [, TAGVALUE])

Where:

TAGNAME	the name of the tag to check on each modifier.
TAGVALUE	the value to match.

Case is ignored as usual, but otherwise the match must be exact.

If no TAGVALUE is given, any value for the TAGNAME counts as a match.

Returns 0 if no matches are found.

This function performs text matching, so 0 matches 0 but **does not** match 0.0 or any other variation that would match numerically.

See also @HASMODWITHTAGVALUE.

@HasModWithTagMin

Searches the item's modifiers for those that match a given value for a given tag and returns the lowest level from those found.

```
@HASMODWITHTAGMIN( TAGNAME [, TAGVALUE ] )
```

Where:

TAGNAME	the name of the tag to check on each modifier.
TAGVALUE	the value to match.

Case is ignored as usual, but otherwise the match must be exact.

If no TAGVALUE is given, any value for the TAGNAME counts as a match.

Returns 0 if no matches are found.

This function performs text matching, so 0 matches 0 but **does not** match 0.0 or any other variation that would match numerically.

See also @HASMODWITHTAGVALUEMIN.

@HasModWithTagValue

Searches the item's modifiers for those that match a given value for a given tag and returns the highest level from those found.

```
@HASMODWITHTAGVALUE( TAGNAME [, TAGVALUE ] )
```

Where:

TAGNAME	the name of the tag to check on each modifier.
TAGVALUE	the value to match.

Case is ignored as usual, but otherwise the match must be exact.

If no TAGVALUE is given, any non-zero value for the TAGNAME counts as a match.

Returns 0 if no matches are found.

This function performs numeric matching, so all values will be converted to numbers before comparisons are made. Tags with non-numerical values will evaluate to 0. This is a direct conversion, not a math-enabled Solver evaluation.

See also @HASMODWITHTAG.

@HasModWithTagValueMin

Searches the item's modifiers for those that match a given value for a given tag and returns the lowest level from those found.

```
@HASMODWITHTAGVALUEMIN( TAGNAME [, TAGVALUE ] )
```

Where:

TAGNAME	the name of the tag to check on each modifier.
TAGVALUE	the value to match.

Case is ignored as usual, but otherwise the match must be exact.

If no TAGVALUE is given, any non-zero value for the TAGNAME counts as a match.

Returns 0 if no matches are found.

This function performs numeric matching, so all values will be converted to numbers before comparisons are made. Tags with non-numerical values will evaluate to 0. This is a direct conversion, not a math-enabled Solver evaluation.

See also @HASMODWITHTAGMIN.

@If

This function returns values depending on evaluation of expressions.

```
@IF(EXPRESSION THEN RESULT [ ELSEIF EXPRESSION THEN RESULT ][ ELSE ALTRESULT ] )
```

Where:

EXPRESSION	is an expression that should result in 0 for FALSE, or another number for TRUE.
RESULT	is the value returned if EXPRESSION is True.
ALTRESULT	is the value returned if no EXPRESSION is found to be True.

Each EXPRESSION is evaluated in turn until a TRUE value is obtained, or until all are exhausted. As soon as a TRUE value is obtained, the corresponding RESULT is returned. If no EXPRESSION values are True, the ALTRESULT value is returned.

As many ELSEIF blocks as desired may be used, but only one ELSE block is allowed.

@IndexedValue

This function allows for returning a value based on the numerical index specified.

> @INDEXEDVALUE(INDEX, VALUE LIST)

Where:

INDEX	is the expression that evaluates to a numerical index into the list of values that follows.
VALUE LIST	is the list of values that are to be returned, separated by commas, and enclosed in quotes or braces as required. Each value may be an expression.

The VALUE LIST can be as long as required, but you should ensure that the INDEX evaluates to a number between 1 and the number of items in the list. If INDEX evaluates to zero, an empty string is returned. If INDEX evaluates to a number greater than the number of items in the list, the last item in the list is returned.

@Int

This function returns the integer portion of a number.

> @INT(VALUE)

Where:

VALUE	is an expression that evaluates to a number.

The difference between @FIX and @INT is in negative numbers: @INT returns the first negative integer less than or equal to VALUE, while @FIX returns the first negative integer greater than or equal to VALUE.

@IsEven

This function returns TRUE if the given value is even.

> @ISEVEN(VALUE)

Where:

VALUE	is an expression that evaluates to a number.

@ItemHasMod

This function is used to determine if the specified trait has a certain modifier applied to it.

> @ITEMHASMOD(TRAITNAME, MODNAME)

Where:

TRAITNAME	is the name of a trait.
MODNAME	is the name of a modifier.

If the specified trait is found on the character, and the specified modifier is applied to that trait, then this function returns the level of the modifier. Otherwise, this function returns 0.

TRAITNAME and MODNAME may be enclosed in quotes or braces, and should be if either might contain a comma.

This function will look for modifiers that have name extensions using either the old NAME (NAME EXT) format, or the newer NAME, NAME EXT format that modifiers display in the UI now.

@ItemsInLibraryGroup

Returns the number of items for a given Group in a character's library.

> @ITEMSINLIBRARYGROUP(GROUPNAME)

This function returns the number of items for the GROUPNAME specified, as found in the character's loaded library's Group collection. This is specifically the Library count, as currently loaded, just as it would be referenced if you were creating a group list for use in a Select dialog.

@ItemsInLibraryList

Returns the number of items for a given List in the character's library.

> @ITEMSINLIBRARYLIST(LISTNAME)

This function returns the number of items for the LISTNAME specified, as found in the character's loaded library's Lists collection.

Be aware that List usage is not the same as Group usage in GCA, as a List is just a collection of text lines, while a Group is specifically a collection of items, one per line. This means that a List may consist of many items per line, and its usage by the file author will likely not correspond to one item per line, but may result in many items in the destination output. Instead of counting the items in the List, you may want to use @ITEMSINLIST to count the items in an equivalently constructed text list instead.

@ItemsInList

Returns the number of items in a text list.

> @ITEMSINLIST([#CODES()] ITEMLIST)

This function allows you to count the items in the given list and return that value. The #CODES() special directive allows you to specify special commands to control how the list is processed.

By default, the list of items is comma separated, and each individual entry may be enclosed in quotes or braces as usual.

The only code currently supported for #CODES() is #CODES(SEP=X) to change the separator from a comma to some other character. NOTE: If you specify multiple characters, each one individually is a separator, not the whole string as a single unit. You may enclose the separator in quotes or braces, such as #codes(sep=" ") to separate on spaces.

@Len

```
@LEN(TEXT)
```

Returns the length of the given TEXT.

@Log

This function returns the Base 10 Log of a number.

```
@LOG(VALUE)
```

Where:

VALUE	is an expression that evaluates to a number.

@Max

This function returns the maximum value from a list of values.

```
@MAX(VALUELIST)
```

Where:

VALUELIST	is the list of values that are to be evaluated, separated by commas. Each value may be an expression.

The VALUELIST can be as long as required.

@Min

This function returns the minimum value from a list of values.

```
@MIN(VALUELIST)
```

Where:

VALUELIST	is the list of values that are to be evaluated, separated by commas. Each value may be an expression.

The VALUELIST can be as long as required.

@Modulo

This function returns the remainder of a value divided by a divisor.

```
@MODULO(VALUE, DIVISOR)
```

Where:

VALUE	is an expression that evaluates to a number.
DIVISOR	is an expression that evaluates to a number.

The VALUE will be divided by the DIVISOR, and the value of the remainder will be returned.

@NLog

This function returns the Natural Log of a number.

> @NLOG(VALUE)

Where:

VALUE	is an expression that evaluates to a number.

You may use @LogN as a synonym for @NLog.

@OwnerHasMod

This function is used to determine if the owner of the containing modifier has a certain modifier applied to it.

> @OWNERHASMOD(ModName)

Where:

ModName	is the name of a modifier.

This function returns the level of the modifier specified, if the modifier is applied to the owner of the containing trait. Otherwise, it returns 0.

Owner in this case refers to the trait or modifier that has applied to it the modifier that includes this function.

This function will look for modifiers that have name extensions using either the old Name (Name Ext) format, or the newer Name, Name Ext format that modifiers display in the UI now.

@Power

This function returns the value of one number raised to the power of another.

> @POWER(VALUE, POWER)

Where:

VALUE	is an expression that evaluates to a number.
POWER	is an expression that evaluates to an integer.

The value returned will be that of VALUE raised to the power of POWER. If POWER doesn't evaluate to an integer, 1 will be returned.

@Round

This function returns a value rounded to the number of decimal places specified.

> @ROUND(VALUE [, PLACES])

Where:

VALUE is an expression that evaluates to a number.

PLACES is an expression that evaluates to an integer.

The value returned will be that of VALUE rounded to the number of decimal places specified by PLACES. If PLACES isn't given, or evaluates to a negative number, VALUE will be rounded to an integer value.

@SameText

This function is used to determine if two text values are the same.

```
@SAMETEXT(VALUE1, VALUE2)
```

Where:

VALUE1 is the first text value.

VALUE2 is the second text value.

If VALUE1 and VALUE2 match, a value of 1 is returned, otherwise 0 is returned. Comparisons, as with all such features in GCA, are case insensitive. Both VALUE1 and VALUE2 may be enclosed in quotes or braces, which will be stripped off before any comparison is made.

@Sqr

This function returns the square root of a number.

```
@SQR(VALUE)
```

Where:

VALUE is an expression that evaluates to a number.

You may use @SQRT as a synonym for @SQR.

@StartsWith

```
@STARTSWITH( DOESTHIS, STARTWITHTHIS )
```

Returns 1 if DOESTHIS starts with STARTWITHTHIS, 0 if not. Case is ignored.

If you want to check for parens ( ), braces { }, or quotes ", you can instead use these variables as needed: %CLOSEPAREN, %OPENPAREN, %CLOSEBRACE, %OPENBRACE, %QUOTES.

Note that the %CLOSEPAREN, %OPENPAREN, %CLOSEBRACE, and %OPENBRACE special variables should not be necessary, if the corresponding characters are enclosed within quotes in the function, such as

```
@startswith(")")
```

The %QUOTES variable is still required, as there is no way to enclose the double quote character safely, but it's also the least likely to be needed for anything.

@SumList

Returns the sum of the values in a comma separated list.

```
@SUMLIST(VALUELIST)
```

Where:

VALUELIST	is the list of values that are to be summed, separated by commas. Each value may be an expression.

The VALUELIST can be as long as required.

@TextIndexedValue

This function returns a value based on a text index item, and a list of (item, value) pairs

```
@TEXTINDEXEDVALUE(INDEX, (ITEM, VALUE) [, (ITEM, VALUE) ] [, ELSE ALTVALUE ] )
```

Where:

INDEX	is the text value that will be compared to each ITEM in the (ITEM, VALUE) pairs to find a match.
ITEM	is the text to be compared to INDEX to find a match.
VALUE	is the expression to be evaluated and returned if a match is found.
ALTVALUE	is the expression to be evaluated and returned if no ITEM is found to match INDEX.

You may include as many (ITEM, VALUE) pairs as you wish. Each pair must be enclosed in parens.

If ITEM and INDEX match, the corresponding VALUE is evaluated and returned. Comparisons, as with all such features in GCA, are case insensitive. If no match is found, ALTVALUE is evaluated and returned, if provided.

@TextIsInList, @TextIsInListAlt

Two similar functions to determine if a given text snippet can be found within a list of text items.

```
@TEXTISINLIST( ISTHIS, { INTHISITEMLIST } )
@TEXTISINLISTALT( ISTHIS, USINGTHISSEPARATORCHARACTER, { INTHISITEMLIST } )
```

Both functions are similar, except that @TEXTISINLIST assumes a comma separated list of items, as is used in the CAT() or PAGE() tags, while @TEXTISINLISTALT allows you to specify the character used for separating the items in the list, such as the pipe (|) character used in SKILLUSED().

You should always enclose the list parameter inside braces to ensure you are safely defining the list of items as a single unit. You can, of course, either specify the list items manually, or obtain them using something like $TextValue(me::cat).

Comparison of items ignores case, as always. The value returned is 0 if the item was not found, or the index number (1-based) of the list item that was matched.

@TextIsInText

@TEXTISINTEXT(ISTHIS, WITHINTHIS)

Returns 1 if the ISTHIS text is contained within the WITHINTHIS text, and 0 if not. As with most such things in GCA, case is ignored.

@TotalChildrenTag

Totals up the values of the specified tag from all children and returns it.

@TOTALCHILDRENTAG(TAG [#NOMULT])

Note that children only are viewed; it does not recurse through nested children. It does, however, multiply the tag value by the child's COUNT to get the total value. If you don't want the value multiplied by COUNT, include the #NOMULT directive in the function call.

@TotalOwnerChildrenTag

Same as @TOTALCHILDRENTAG but meant to be used from a modifier, to get the owning trait's children.

@TOTALOWNERCHILDRENTAG(TAG [#NOMULT])

Special Case Substitutions

There are several special variables that may be used in math-enabled areas. These variables are replaced before any expression is evaluated, so they may be used safely in math expressions.

%LEVEL

This variable will be replaced by the level of the containing item.

%COUNT

This variable will be replaced by the value of the COUNT() tag of the containing item.

$TEXTVALUE

This variable functions very much like a text function. This is a means of inserting a bit of text, based on a trait value, into an expression before it is evaluated.

$TEXTVALUE(KEYWORD[::TAG])

Where:

KEYWORD	is a trait name, or one of a variety of keywords:

CHAR	references the character.
OWNER	references the trait or modifier that owns the containing item.
ME	references the containing item.
DEFAULT	references the trait being defaulted from.

PREREQ	returns a list of values for the traits in the pre-requisite list.
LOWPREREQ	returns a list of values for the traits in the pre-requisite list.

TAG	is a tag reference, if needed.

If TAG is used, a tag value may be obtained from a trait, from the character, from the owning item, or from the containing item itself.

$VAL

This is identical in function to $TEXTVALUE(), just shorter.

TEXT PROCESSING

In addition to the ability to evaluate math expressions and handle various math functions in service to that, GCA has various text processing functions as well. The text functions are available in math enabled places (in addition to the standard math functions), to enable processing of text features that may help create the math expressions to be evaluated. Text functions are also available in the processing of a number of tags, sometimes instead of the more full featured math processing.

Text Functions

Text functions always begin with the $ character. When GCA sees a $ character in math or text processing enabled areas, it will look for and evaluate any text functions.

As with math functions, the entire text function will be replaced by the returned value.

$BonusStrings

```
$BONUSSTRINGS(TAGNAME)
```

Returns the concatenated string of all text bonuses targeted to the given tag of the calculating trait, or an empty string if there are none.

$Eval

This function simply allows for processing a math expression within the text solver.

```
$EVAL(EXPRESSION)
```

Where:

EXPRESSION	is the expression sent to the math solver.

You may use $EVALUATE or $SOLVER as synonyms for $EVAL.

$if

This function allows for evaluating expressions to determine which of many options will be returned.

```
$IF( EXPRESSION THEN RESULT [ ELSEIF EXPRESSION THEN RESULT [ ... ] ] [ ELSE ALTRESULT ] )
```

Where:

EXPRESSION	is a math expression that should result in 0 for FALSE, or another number for TRUE.
RESULT	is the text returned if EXPRESSION evaluates to TRUE.
ALTRESULT	is the text returned if EXPRESSION evaluates to FALSE.

Each EXPRESSION is evaluated in turn until a TRUE value is obtained, or until all are exhausted. As soon as a TRUE value is obtained, the corresponding RESULT is returned. If no EXPRESSION values are True, the ALTRESULT value is returned.

As many ELSEIF blocks as desired may be used, but only one ELSE block is allowed.

RESULT and ALTRESULT may be enclosed in quotes or braces, and should be if either might contain a keyword.

$IndexedValue

This function allows for returning a text value based on the numerical index specified.

```
$INDEXEDVALUE(INDEX, VALUELIST)
```

Where:

INDEX	is the expression that evaluates to a numerical index into the list of text values that follows.
VALUELIST	is the list of text values that are to be returned, separated by commas, and enclosed in quotes or braces as required.

The VALUELIST can be as long as required, but you should ensure that the INDEX evaluates to a number between 1 and the number of items in the list. If INDEX evaluates to zero, an empty string is returned. If INDEX evaluates to a number greater than the number of items in the list, the last item in the list is returned.

$InsertInto

Inserts some text into some other text.

```
$INSERTINTO( INTOTHIS, PUTTHIS, ATPOSITION )
```

INTOTHIS is the text that is going to receive the PUTTHIS, and the text is inserted at position ATPOSITION.

$ListNoBlanks

Processes a list of items and removes the empty entries.

```
$LISTNOBLANKS( [ #CODES() ] ITEMLIST )
```

This function allows you to process a list of items and remove the empty entries. The #codes() special directive allows you to specify special commands to control how the list is processed.

By default, the list of items is comma separated, and each individual entry may be enclosed in quotes or braces as usual.

You may specify #CODES(HASPREFIX) or #CODES(PREFIXTAGS) to tell the function that the entries in the list are expected to be trait names tagged with prefix tags and therefore any item that does **not** have both **or** where either part is not specified, are considered blank items.

You may specify #CODES(SEP=X) to change the separator from a comma to some other character. NOTE: If you specify multiple characters, each one individually is a separator, not the whole string as a single unit. You may enclose the separator in quotes or braces, such as #codes(sep=" ") to separate on spaces.

If you specify multiple codes entries, separate them with commas.

$Modifiers

Returns a list of the modifiers applied to the trait.

> $MODIFIERS([CAPTIONS | FULLNAMES] [, VALUES] [, NOPARENS])

Where:

CAPTIONS	tells the function to return the usual captions (also known as shortnames and defined by a modifier's SHORTNAMES() tag).
FULLNAMES	tells the function to return the full names of the modifiers.
VALUES	tells the function to include the modifiers' values.
NOPARENS	tells the function not to enclose the return text in parentheses.

Use $MODIFIERS() without parameters to get the usual captions (shortnames), but without values, enclosed in parentheses. Use a comma separated list of parameters listed above to modify this behavior

Note that if there are no modifiers for the trait, the return value will be empty, not a set of empty parentheses.

For example,

> $modifiers(values, fullnames)

will return the full names of the modifiers along with their values.

This is probably only useful with DISPLAYNAMEFORMULA(), but that's why I added it.

$TextIndexedValue

This function returns a text value based on a text index item, and a list of (item, value) text pairs

> $TEXTINDEXEDVALUE(INDEX, (ITEM, VALUE) [, (ITEM, VALUE)] [, ELSE ALTVALUE])

Where:

INDEX	is the text value that will be compared to each ITEM in the (ITEM, VALUE) pairs to find a match.
ITEM	is the text to be compared to INDEX to find a match.
VALUE	is the text to be returned if a match is found.

ALTVALUE	is the text to be returned if no ITEM is found to match INDEX.

You may include as many (ITEM, VALUE) pairs as you wish. Each pair must be enclosed in parens.

If ITEM and INDEX match, the corresponding VALUE is returned. Comparisons, as with all such features in GCA, are case insensitive. If no match is found, ALTVALUE is returned, if provided.

Rolling

To support random templates and randomized characters, GCA includes some special functions for rolling and retrieving rolled results. Those are detailed in this special section because they include a mix of numeric and text functions, and I thought it was better to cover them together.

Generic Roll Functions

@Random

The @RANDOM() function returns a random integer value. Note that the parameter values are applied differently depending on whether one or two values are supplied.

> @RANDOM(MAX)
> @RANDOM(MIN, MAX)

Where:

MAX	is an integer value.
MIN	is an integer value.

If only MAX is specified, then the value returned ranges from 1 to the MAX value (inclusive). If both MIN and MAX are specified, then the value returned ranges from the MIN to the MAX (inclusive).

@Roll

The @ROLL() function returns the sum total for a series of random results simulating rolled dice.

> @ROLL(NUMBER)
> @ROLL(NUMBER, SIDES)

Where:

NUMBER	is an integer value representing the number of dice to roll.
SIDES	is an integer value representing the number of sides per die. (Assumes a die numbered 1 to SIDES.)

If SIDES is not specified it uses six-sided dice. So using @roll(2,4) would 'roll' 2 four-sided dice, and would return the total sum of all 'rolled' values.

$RollValues

The $ROLLVALUES() function returns a text value that is a sequence of values separated by + signs, representing every roll made for the given parameters

> $ROLLVALUES(NUMBER)
> $ROLLVALUES(NUMBER, SIDES)

Where:

NUMBER	is an integer value representing the number of dice to roll.
SIDES	is an integer value representing the number of sides per die. (Assumes a die numbered 1 to SIDES.)

If SIDES is not specified it uses six-sided dice. So using $rollvalues(2,4) would 'roll' 2 four-sided dice, and would return the value as something similar to 1+4, so that it can be included in a math expression.

Generic Stored Result Functions

GCA will remember the last value returned for the @ROLL, @RANDOM, and $ROLLVALUES functions. Bear in mind that these saved results will change each time these functions are called, so they're only really useful inside the same operation, such as in an @IF, and only reliable even then if you only use one of that function in the evaluation.

Note that these saved values are universal to GCA and **not** to the character or the trait where they are used.

To get the last returned value, use one of these functions. Note that they don't have parameters, but you still must include the ().

@LastRoll

Returns the last value returned by the @ROLL() function.

```
@LASTROLL()
```

@LastRandom

Returns the last value returned by the @RANDOM() function.

```
@LASTRANDOM()
```

$LastRollValues

Returns the last value returned by the $ROLLVALUES() function.

```
@LASTROLLVALUES()
```

Named Roll Functions

These functions each work the same as their unnamed counterparts. The only difference is that you can name the roll by specifying the name as the first parameter. By using these functions and specifying unique names, you can retrieve them later by name and be less concerned that another use of the unnamed functions will overwrite the values.

These named versions **do not** place their results into the LASTX values used by the generic functions, so you can't retrieve a value generated here with the LASTX() functions (but you can with the LASTNAMEDX() functions and a valid name). This allows you to use named results for longer storage and unnamed ones for transitory values without one messing up the other.

Note that all of these named results and their stored values are universal to GCA, **not** to the character or the trait where they are used.

@NamedRandom

The @NAMEDRANDOM() function returns a random integer value. Note that the parameter values are applied differently depending on whether one or two values are supplied after the NAME.

```
@NAMEDRANDOM( NAME, MAX )
@NAMEDRANDOM( NAME, MIN, MAX )
```

Where:

NAME	the name of the stored result.
MAX	is an integer value.
MIN	is an integer value.

If only MAX is specified, then the value returned ranges from 1 to the MAX value (inclusive). If both MIN and MAX are specified, then the value returned ranges from the MIN to the MAX (inclusive).

@NamedRoll

The @NAMEDROLL() function returns the sum total for a series of random results simulating rolled dice.

```
@NAMEDROLL( NAME, NUMBER )
@NAMEDROLL( NAME, NUMBER, SIDES )
```

Where:

NAME	the name of the stored result.
NUMBER	is an integer value representing the number of dice to roll.
SIDES	is an integer value representing the number of sides per die. (Assumes a die numbered 1 to SIDES.)

If SIDES is not specified it uses six-sided dice. So using @namedroll(Bob, 2, 4) would 'roll' 2 four-sided dice, and would return the total sum of all 'rolled' values, as well as storing it with the name of Bob.

$NamedRollValues

The $NAMEDROLLVALUES() function returns a text value that is a sequence of values separated by + signs, representing every roll made for the given parameters

```
$NAMEDROLLVALUES( NAME, NUMBER )
$NAMEDROLLVALUES( NAME, NUMBER, SIDES )
```

Where:

NAME	the name of the stored result.
NUMBER	is an integer value representing the number of dice to roll.
SIDES	is an integer value representing the number of sides per die. (Assumes a die numbered 1 to SIDES.)

If SIDES is not specified it uses six-sided dice. So using $namedrollvalues(Bob, 2, 4) would 'roll' 2 four-sided dice, and would return the value as something similar to 1+4, so that it can be included in a math expression, as well as storing it with the name of Bob.

Named Stored Result Functions

These functions allow you retrieve the named results that were stored by using one of the named roll functions. In this way, you can roll using a named function, and use that value both where it was rolled and elsewhere later.

@LastNamedRoll

Returns the last stored/returned result for the given name. Returns 0 if there is no such name.

```
@LASTNAMEDROLL( NAME )
```

Where:

NAME	the name of the stored result to retrieve.

If there is no stored value for NAME, 0 is returned.

@LastNamedRandom

Returns the last stored/returned result for the given name. Returns 0 if there is no such name.

```
@LASTNAMEDRANDOM( NAME )
```

Where:

NAME	the name of the stored result to retrieve.

If there is no stored value for NAME, 0 is returned.

$LastNamedRollValues

Returns the last stored/returned result for the given name. Returns "0" if there is no such name.

```
$LASTNAMEDROLLVALUES( NAME )
```

Where:

NAME	the name of the stored result to retrieve.

If there is no stored value for NAME, 0 is returned.

DIRECTIVES

This section will cover what you need to know to use directives in trait definitions. Directives are like commands that tell GCA to do particular things when a trait is added to a character. Until a trait is added to the character, a directive does nothing.

GCA will remove directives from the trait data when the trait is added to the character and the directives are processed.

As a general rule, directives should be placed in the x() tag, which is specifically used to contain extended processing information, such as directives. Some directives, by nature, may not be used in the x() tag, and may instead be included elsewhere in the trait definition. Also, directives with result variables will have those result variables used wherever necessary, and not contained in the x() tag.

#BuildCharItemList

The #BUILDCHARITEMLIST directive works using existing character data, rather than items from the data files, to create a list of items. Note that this is constructed when the trait is added to the character, so unlike #BUILDSELECTLIST, which does something similar (and has the same construction), the list will not contain items added after this item was added by the user.

> #BUILDCHARITEMLIST(TYPE WHERE TAG [IS | ISNOT | INCLUDES | EXCLUDES | LISTINCLUDES | LISTEEXCLUDES] TAGVALUE , TEMPLATE([SOME TEXT] %LISTITEM% [MORE TEXT]))

#BUILDCHARITEMLIST allows for creating a list of items that are built based on the specified TEMPLATE() text, where each occurrence of the %LISTITEM% variable is replaced by the full name of any traits selected based on the WHERE statement that makes up the primary portion of this tag. Should no TEMPLATE() be included, then a list of the item names is returned.

The WHERE clause works like this:

TYPE	should be replaced with the type of traits to be looked at. This is a keyword or a prefix tag, such as Skills or SK:.
TAG	should be replaced with the trait tag whose values we want to examine for our selection process. This is just the tag name, such as cat or tl.

[IS | ISNOT | INCLUDES | EXCLUDES | LISTINCLUDES | LISTEEXCLUDES] Select one of these comparison options. Be sure whichever one you use is **all one word**, or the #BUILDCHARITEMLIST will be rejected, and your template will fail.

- IS means the TAG and the TAGVALUE must match.
- ISNOT means the TAG and the TAGVALUE must not match.
- INCLUDES means the TAG must include, anywhere, the TAGVALUE
- EXCLUDES means the TAG must not include, anywhere, the TAGVALUE
- LISTINCLUDES means the TAG must have somewhere in its list of values the TAGVALUE. So you could specify a TAG of cat and a TAGVALUE of Hobbies, and as long as one of the CAT() categories for an item is Hobbies, it would be selected.

- LISTEXCLUDES means the TAG must not have anywhere in its list of values the TAGVALUE. So, using a TAG of cat and a TAGVALUE of Hobbies, any trait that includes Hobbies in the CAT() would be excluded from being in the result list.

TAGVALUE should be replaced with the value of the tag that should be looked at in the comparison. Include quotes or braces around this value if it includes spaces or commas, or just in general for safety.

And the template:

TEMPLATE([SOME TEXT] %LISTITEM% [MORE TEXT]) Allows you to specify the appearance of the text for each item created in the output list. Every item in the list will be output to match the template, with the special variable %LISTITEM% being replaced by the item from the list that's currently being output. Text to the left and right of the %LISTITEM% will be used exactly as is to create the output item, so any quotes or other special characters will be used as is, and will appear in the output list. You must be careful, however, not to include single, unmatched parens, as that will mess up GCA's ability to parse things correctly.

#BuildCombo

The #BUILDCOMBO directive prompts GCA to offer the user the Combination Editor window, with which they can build the combination they intend to use.

> #BUILDCOMBO([2 | 3])

The two available options are #BUILDCOMBO(2) or #BUILDCOMBO(3), which will allow for building 2 attack or 3 attack combinations, respectively.

#BuildIt

From GCA wiki, by Armin; modified.

The #BUILDIT directive prompts GCA to offer the user the Modifiers window, with which they can build the version of the added item they intend to use.

> #BUILDIT

This directive has no options, and therefore has no parentheses or available tags.

#BuildLibraryItemList

The #BUILDLIBRARYITEMLIST directive works on the character's Library data to create a list of items based on tag values.

> #BUILDLIBRARYITEMLIST (TYPE WHERE TAG [IS | ISNOT | INCLUDES | EXCLUDES | LISTINCLUDES |
> LISTEEXCLUDES] TAGVALUE , TEMPLATE([SOME TEXT] %LISTITEM% [MORE TEXT]))

This works in the same was as #BUILDCHARITEMLIST, but on Library data. Bear in mind that no tags applied to character items will be the same in the Library items: that's just book data straight from the files.

#BUILDLIBRARYITEMLIST allows for creating a list of items that are built based on the specified TEMPLATE() text, where each occurrence of the %LISTITEM% variable is replaced by the full name of any traits selected based on the WHERE statement that makes up the primary portion of this tag. Should no TEMPLATE() be included, then a list of the item names is returned.

The WHERE clause works like this:

TYPE	should be replaced with the type of traits to be looked at. This is a keyword or a prefix tag, such as Skills or SK:.
TAG	should be replaced with the trait tag whose values we want to examine for our selection process. This is just the tag name, such as cat or tl.

[IS | ISNOT | INCLUDES | EXCLUDES | LISTINCLUDES | LISTEEXCLUDES] Select one of these comparison options. Be sure whichever one you use is **all one word**, or the #BUILDLIBRARYITEMLIST will be rejected, and your template will fail.

- IS means the TAG and the TAGVALUE must match.
- ISNOT means the TAG and the TAGVALUE must not match.
- INCLUDES means the TAG must include, anywhere, the TAGVALUE
- EXCLUDES means the TAG must not include, anywhere, the TAGVALUE
- LISTINCLUDES means the TAG must have somewhere in its list of values the TAGVALUE. So you could specify a TAG of cat and a TAGVALUE of Hobbies, and as long as one of the CAT() categories for an item is Hobbies, it would be selected.
- LISTEXCLUDES means the TAG must not have anywhere in its list of values the TAGVALUE. So, using a TAG of cat and a TAGVALUE of Hobbies, any trait that includes Hobbies in the CAT() would be excluded from being in the result list.

TAGVALUE	should be replaced with the value of the tag that should be looked at in the comparison. Include quotes or braces around this value if it includes spaces or commas, or just in general for safety.

And the template:

TEMPLATE([SOME TEXT] %LISTITEM% [MORE TEXT]) Allows you to specify the appearance of the text for each item created in the output list. Every item in the list will be output to match the template, with the special variable %LISTITEM% being replaced by the item from the list that's currently being output. Text to the left and right of the %LISTITEM% will be used exactly as is to create the output item, so any quotes or other special characters will be used as is, and will appear in the output list. You must be careful, however, not to include single, unmatched parens, as that will mess up GCA's ability to parse things correctly.

#BuildList

From GCA wiki, by Armin; modified.

The #BUILDLIST directive allows you to create a list with custom text options, building up from a list output by another directive or entered directly.

```
#BUILDLIST( LIST(ITEMLIST), TEMPLATE( [SOME TEXT] %LISTITEM% [MORE TEXT] ) )
```

LIST(ITEMLIST) Allows you to specify the comma-separated list of items to be used when building the output list. Any item that includes commas must be enclosed within quotes or curly {} braces. If an item includes quotes, you should enclose it in braces instead of more quotes. If you use braces, GCA will not attempt to also remove any quotes or doubled quotes.

TEMPLATE([SOME TEXT] %LISTITEM% [MORE TEXT]) Allows you to specify the appearance of the text for each item created in the output list. Every item in the list will be output to match the template, with the special variable %LISTITEM% being replaced by the item from the list that's currently being output. Text to the left and right of the %LISTITEM% will be used exactly as is to create the output item, so any quotes or other special characters will be used as is, and will appear in the output list. You must be careful, however, not to include single, unmatched parens, as that will mess up GCA's ability to parse things correctly.

You may use more than one instance of the %LISTITEM% variable in the output template if needed.

This example:

```
#buildlist(list(Hearing, Taste and Smell, Touch, Vision), template(AD:Acute %ListItem%=2))
```

will output this list:

```
AD:Acute Hearing=2, AD:Acute Taste and Smell=2, AD:Acute Touch=2, AD:Acute Vision=2
```

in place of the #BUILDLIST directive.

#BuildSelectList

This is a special SELECT()/SELECTX() specific directive to allow for building a list of items for use in the SELECT() item list, which is processed when the SELECT() becomes active. This means it may possibly include items from earlier SELECT() picks, which wouldn't be the case for normal directives, which are processed when the trait is added to the character.

```
#BUILDSELECTLIST( TYPE WHERE TAG [ IS | ISNOT | INCLUDES | EXCLUDES | LISTINCLUDES | LISTEXCLUDES ]
TAGVALUE , TEMPLATE( [SOME TEXT] %LISTITEM% [MORE TEXT] ) )
```

#BUILDSELECTLIST allows for creating a list of items that are built based on the specified TEMPLATE() text, where each occurrence of the %LISTITEM% variable is replaced by the full name of any traits selected based on the where statement that makes up the primary portion of this tag. Should no TEMPLATE() be included, then a list of the item names is returned.

The WHERE clause works like this:

TYPE	should be replaced with the type of traits to be looked at. This is a keyword or a prefix tag, such as Skills or SK:.
TAG	should be replaced with the trait tag whose values we want to examine for our selection process. This is just the tag name, such as cat or tl.

[IS | ISNOT | INCLUDES | EXCLUDES | LISTINCLUDES | LISTEEXCLUDES] Select one of these comparison options. Be sure whichever one you use is **all one word**, or the #BUILDSELECTLIST will be rejected, and your template will fail.

IS means the TAG and the TAGVALUE must match.

ISNOT means the TAG and the TAGVALUE must not match.

INCLUDES means the TAG must include, anywhere, the TAGVALUE

EXCLUDES means the TAG must not include, anywhere, the TAGVALUE

LISTINCLUDES means the TAG must have somewhere in its list of values the TAGVALUE. So you could specify a TAG of cat and a TAGVALUE of Hobbies, and as long as one of the CAT() categories for an item is Hobbies, it would be selected.

LISTEXCLUDES means the TAG must not have anywhere in its list of values the TAGVALUE. So, using a TAG of cat and a TAGVALUE of Hobbies, any trait that includes Hobbies in the CAT() would be excluded from being in the result list.

TAGVALUE should be replaced with the value of the tag that should be looked at in the comparison. Include quotes or braces around this value if it includes spaces or commas, or just in general for safety.

And the template:

TEMPLATE([SOME TEXT] %LISTITEM% [MORE TEXT]) Allows you to specify the appearance of the text for each item created in the output list. Every item in the list will be output to match the template, with the special variable %LISTITEM% being replaced by the item from the list that's currently being output. Text to the left and right of the %LISTITEM% will be used exactly as is to create the output item, so any quotes or other special characters will be used as is, and will appear in the output list. You must be careful, however, not to include single, unmatched parens, as that will mess up GCA's ability to parse things correctly.

Let's look at an example:

```
select(_
   text("Please select an appropriate number or value of these traits."),
   pointswanted(exactly 4),
   itemswanted(exactly 1),
   list(_
     #BuildSelectList(Skills where cat includes "Melee Combat",
             template(_
             #newitem(SK:Increase %ListItem% by 4 points,
                     existing(SK:%ListItem%)) #codes(upto %points+4pts, downto %points+4pts)_
                   )_
              )_
   )_
)
```

In this SELECT(), we want to build a list of skills where some portion of the CAT() tag includes the phrase melee combat. For every such skill on the character, a #NEWITEM() will be generated as a member of the LIST() tag for the SELECT(), and that item will allow for an existing skill to be increased by 4 points, and only 4 points.

So, if the character had two skills from the "Combat/Weapons - Melee Combat" category of skills, let's say Broadsword and Axe/Mace, the result of the SELECT() statement above would look like this immediately before it was processed by the selection window, as if you'd written it this way:

```
select(_
    text("Please select an appropriate number or value of these traits."),
    pointswanted(exactly 4),
    itemswanted(exactly 1),
    list(_
       #newitem(SK:Increase Broadsword by 4 points,
                  existing(SK:Broadsword)) #codes(upto %points+4pts, downto %points+4pts),
       #newitem(SK:Increase Axe/Mace by 4 points,
                  existing(SK:Axe/Mace)) #codes(upto %points+4pts, downto %points+4pts)_
    )_
)
```

Note: GCA no longer requires you to use #NEWITEM() and EXISTING() to access existing traits. You can get the same results now with less code. All things from the example would be the same except for the TEMPLATE(), which could be simplified like this:

```
template(SK:%ListItem% #codes(upto %points+4pts, downto %points+4pts) )_
```

Would get us this:

```
select(_
    text("Please select an appropriate number or value of these traits."),
    pointswanted(exactly 4),
    itemswanted(exactly 1),
    list(_
       SK:Broadsword #codes(upto %points+4pts, downto %points+4pts),
       SK:Axe/Mace #codes(upto %points+4pts, downto %points+4pts)_
    )_
)
```

The result to the user would be the same, but perhaps a bit less confusing overall: they'd see that the Available list included some traits that they already had (because they're flagged now), and adding them to the Selected list would adjust them by 4pts as expected.

#Choice

From GCA wiki, by Armin; modified.

The #CHOICE directive allows you to provide the user with a list of options, and to obtain the one they pick. This is intended to be a quicker means of getting a simple, single choice from the user than #CHOICELIST.

```
#CHOICE( "ITEM"[=COST] [, "ITEM"[=COST] ] )
```

This directive takes a comma separated list of items. If you also want to provide the user with costs for the items, you need to use the optional cost assignment, which is done by including the cost for each item after an equals sign following the item text.

The costs are not used for making the selection; they would be included for reference only, or if you needed the cost for the selected item later. Only a single item can be selected by the user.

The items in the list may be surrounded by quotes, as shown in the template, or by curly {} braces. You must use one or the other if the item includes commas or equals signs. Use the curly braces instead of the quotes if the item contains quotes, or even if you prefer them. Note that if you do use the braces, GCA will not attempt to remove any quotes or doubled-quotes from the item names. Remember that in either case, the cost assignment would come outside the quotes or braces.

Result Variables

Result variables are replaced by the selection made by the user. Result variables may appear anywhere in the trait definition.

%CHOICE%	GCA will insert the user's selected option in place of this variable.
%CHOICECOST%	GCA will insert the specified cost of the user's selected option in place of this variable.

One or both of the result variables may be used, and each may be used more than once. GCA will make the replacements anywhere the result variables are found in the trait definition. Note that the results placed into the variables by GCA will not include quotes.

Example:

```
Swears oddly (%choice%), -1, x(#Choice("Fracking", "Frelling", "Gorram"))
```

The user would be presented with a Choose Items dialog that asks them to pick one of the items listed. It's the same dialog #CHOICELIST pops up, but without the ability to specify the various text prompts. And it will always be "Want 1". The user checks the box for their selection, clicks OK, and GCA will remove the entire #CHOICE() section from the X() tag contents, leaving their selected option in place of %CHOICE%; in this case, perhaps leaving a quirk of Swears oddly (Gorram) on the character.

Note: As with other directives, GCA will remove this directive from the trait when it is added to the character. Unlike with other directives, the output generated by this directive will be inserted in place of the result variables, instead of in place of this directive.

The Difference Between #Choice/#ChoiceList and SelectX()

Many people are confused by the difference between #CHOICE/#CHOICELIST and the SELECTX() tags. The important difference to remember is that the #CHOICE/#CHOICELIST directives deal with *text*, while SELECTX() deals with *traits*. The results placed into the result variables by #CHOICE/#CHOICELIST are just text values, exactly as specified in the directive; the result of the SELECTX() selections are traits added to the character.

#ChoiceList

The #CHOICELIST directive allows you to provide the user with a list of options, and to obtain the ones they pick.

```
#CHOICELIST( LIST(ITEMLIST),
        NAME(CHOICENAME),
        TITLE(DIALOGTITLE),
        TEXT(DIALOGTEXT),
        ALIASLIST(ALIASLIST),
        ALTXLIST(ALTXLIST),
        DEFAULT(INITIALSELECTIONSLIST),
        AUTOACCEPT(ACCEPT),
        PICKSALLOWED( [ EXACTLY | UPTO | ATLEAST ] NUMBER [, [ EXACTLY | UPTO | ATLEAST ] NUMBER ] ),
        TOTALCOST( [ EXACTLY | UPTO | ATLEAST ] COST),
        METHOD( [ BYNUMBER | BYNUM | BYCOST | BYBOTH ])  )
```

Since this is a tag list, most of the tags are optional.

LIST(ITEMLIST) allows you to specify the list of items to display to the user. If you also want to provide the user with costs for the items, you need to use the optional cost assignment, which is done by including the cost for each item after an equals sign following the item text:

```
LIST( "ITEM"[=COST ] [, "ITEM"[=COST ] ] ).
```

The items in the list may be surrounded by quotes, as shown in the template, or by curly {} braces. You must use one or the other if the item includes commas or equals signs. Use the curly braces instead of the quotes if the item contains quotes, or even if you prefer them. Note that if you do use the braces, GCA will not attempt to remove any quotes or doubled-quotes from the item names. Remember that in either case, the cost assignment would come outside the quotes or braces.

NAME(CHOICENAME) allows you to specify the name of this particular #CHOICELIST. This allows for named result variables, which means multiple #CHOICELIST directives may be used in the same text. If no NAME() is specified, choice will be used.

TITLE(DIALOGTITLE) allows you to specify the title, or caption, of the dialog displayed to the user.

TEXT(DIALOGTEXT) allows you to specify the text explaining the purpose of the dialog displayed to the user.

ALIASLIST(ALIASLIST) allows you to specify a list of text that parallels the LIST() items, but contains alternate text for each option. The number of items specified must match the number of items in LIST(), or GCA will explode. You may not specify costs.

ALTXLIST(ALTXLIST) allows you to specify up to 9 alternate lists, beyond the ALIASLIST(), that also parallel the LIST() items, but likewise contain alternate text for each option. The number of items specified must match the number of items in LIST(), or GCA will explode. You may not specify costs.

Each alternate list is specified with a number, from 1 to 9, in place of the X in the tag name. For example, alt1list() or alt2list().

DEFAULT(INITIALSELECTIONSLIST) allows you to specify a comma separated list of the initially selected items. Each selected item is represented by an integer corresponding to its location in the LIST(), as written in the data file. The first item is 1, the second 2, and so on. Do not specify the items as they appear listed to the user, as that order may not match the order specified in the LIST().

AUTOACCEPT(ACCEPT) If ACCEPT is anything other than empty, GCA will simply accept the default values for the #CHOICELIST without bothering to show the dialog to the user.

PICKSALLOWED([EXACTLY | UPTO | ATLEAST] NUMBER [, [EXACTLY | UPTO | ATLEAST] NUMBER]) allows you to specify what number of items the user is supposed to select. You may specify just the NUMBER by itself, or you may use an optional qualifier as shown. If you use the UPTO or ATLEAST qualifier, you may specify a second number of picks, with a second qualifier, to limit the range introduced. Separate the first block from the second with a comma. Math-enabled; you should enclose the formulas in quotes or braces to protect parsing.

For example, if you want the user to pick at least one option, but they may pick as many as five, you'd use picksallowed(atleast 1, upto 5).

If no qualifier is specified, EXACTLY is assumed.

TOTALCOST([EXACTLY | UPTO | ATLEAST] COST) allows you to specify that you want some total cost, using the costs provided in the LIST(), to be required. You may use the optional qualifiers to provide more leeway in selection. If no qualifier is specified, EXACTLY is assumed. Math-enabled; you should enclose the formulas in quotes or braces to protect parsing.

METHOD([BYNUMBER | BYNUM | BYCOST | BYBOTH]) allows you to specify the method by which GCA should determine if the user has selected the desired number of items. Using BYNUMBER or BYNUM specifies that only the number of items, as specified in

PICKSALLOWED(), is important. Using BYCOST specifies that only the total cost, as specified in TOTALCOST(), is important. Using BYBOTH requires satisfying both the specified number and cost of selected items.

Result Variables

Result variables are replaced by selections made by the user, or sometimes the selections not made. Result variables may appear anywhere in the trait definition.

%CHOICENAME% GCA will insert the user's first selected option in place of this variable.

%CHOICENAMECOST% GCA will insert the specified cost of the user's first selected option in place of this variable.

%CHOICENAMEALIAS% GCA will insert the ALIASLIST() item corresponding to the user's first selected option in place of this variable.

%CHOICENAME#% Replacing # with the numbers from 1 to the number of choices made by the user, GCA will replace this variable with the corresponding LIST() item.

%CHOICENAMECOST#% Replacing # with the numbers from 1 to the number of choices made by the user, GCA will replace this variable with the cost of the corresponding LIST() item.

%CHOICENAME#ALIAS% Replacing # with the numbers from 1 to the number of choices made by the user, GCA will replace this variable with the corresponding ALIASLIST() item.

%CHOICENAME#ALTX% Replacing # with the numbers from 1 to the number of choices made by the user, GCA will replace this variable with the corresponding ALTXLIST() item.

%CHOICENAMELIST% GCA will insert a comma separated list of all the user’s selected options in place of this variable.

%CHOICENAMECOSTLIST% GCA will insert a comma separated list of all the costs corresponding to the user’s selected options, in place of this variable.

%CHOICENAMEALIASLIST% GCA will insert a comma separated list of all the ALIASLIST() items corresponding to the user’s selected options, in place of this variable.

%CHOICENAMEALTXLIST% GCA will insert a comma separated list of all the ALTXLIST() items corresponding to the user’s selected options, in place of this variable.

%CHOICENAMENOTLIST% GCA will insert a comma separated list of all the LIST() items *not* chosen by the user, in place of this variable.

%CHOICENAMECOSTNOTLIST% GCA will insert a comma separated list of all the costs corresponding to items *not* chosen by the user, in place of this variable.

%CHOICENAMEALIASNOTLIST% GCA will insert a comma separated list of all the ALIASLIST() items corresponding to items *not* chosen by the user, in place of this variable.

Any or all of the result variables may be used, and each may be used more than once. GCA will make the replacements anywhere the result variables are found in the trait definition. Note that the results placed into the variables by GCA will not include quotes.

Note: As with other directives, GCA will remove this directive from the trait when it is added to the character. Unlike with other directives, the output generated by this directive will be inserted in place of the result variables, instead of in place of this directive.

The Difference Between #Choice/#ChoiceList and SelectX()

Many people are confused by the difference between #CHOICE/#CHOICELIST and the SELECTX() tags. The important difference to remember is that the #CHOICE/#CHOICELIST directives deal with *text*, while SELECTX() deals with *traits*. The results placed into the result variables by #CHOICE/#CHOICELIST are just text values, exactly as specified in the directive; the result of the SELECTX() selections are traits added to the character.

#DeleteMe

Instructs GCA to delete the item once it's been fully processed.

This may seem a bit odd, but it allows for creating templates that add things to the character and are then removed, so that the user doesn't have to manually remove them later when they no longer serve any purpose.

Note some users may find it confusing if a trait doesn't appear in the Character list after they've added it, so this directive is mostly for those that want to use it in their custom data, or when it might be helpful to use a virtual mini-lens that could simplify SELECT() options without also cluttering up the user's character.

#Edit

The #EDIT directive prompts GCA to offer the user the Edit window, with which they can immediately make adjustments to the trait.

This directive has no options, and therefore has no parentheses or available tags.

#Format

Modifiers now support a new directive, #format, within the shortname() tag, which tells it to format the output in a particular way—pretty much just like displaynameformula(), but obviously set up a tad differently.

```
SHORTNAME( #FORMAT EXPRESSION )
```

For example:

```
shortname( #format $val(me::levelname) )
```

#FORMAT must be the first part of the tag value, and everything that follows is the EXPRESSION to use. If you need spaces on one end or another, enclose the text after the #FORMAT directive in quotes or braces.

Combined with DISPLAYNAMEFORMULA(), this means that you can now usefully create all-in-one modifiers, and don't need to create a separate modifier for each different level of effect. For example:

```
[MODIFIERS]
<Self-Control>
Self-Control Roll, *0.5/*1/*1.5/*2, upto(4), downto(1), group(Self-Control), page(B121),
        shortname(#format $val(me::shortlevelname) ),
        levelnames("15 or less, almost all the time",
                            "12 or less, quite often",
                            "9 or less, fairly often",
                            "6 or less, quite rarely"),
        shortlevelnames(15,
                            12,
                            9,
                            6),
        displaynameformula( You resist on a roll of $val(me::levelname) )
```

(I set up this example in order of increasing costs, which is in decreasing order of Roll value. It could, of course, also be set up for the reverse order, instead.)

Inside GCA:

- the user sees **Self-Control Roll** in the Available Modifiers list
- when added to the character they see **You resist on a roll of 15 or less, almost all the time** in the Applied Modifiers list
- and for the caption within the item using this modifier, they see **15, *0.5**.

If the user then increments the modifier, the values change appropriately to the next levels.

Further, if you need some item to check for the existence of this modifier with @HASMOD(), you can simply use @HasMod(Self-Control Roll), rather than needing to check for the name of each individual level, as is the case now with each level being a separate modifier.

Other examples:

```
[MODIFIERS]
<Chronic Pain>
Interval, *0.5/*1/*1.5/*2, upto(4), downto(1), shortname( #format $val(me::levelname) ),
        levelnames(1 hour, 2 hours, 4 hours, 8 hours), group(Chronic Pain), page(B126),
        displaynameformula( $val(me::name): $val(me::levelname) )

Frequency, *0.5/*1/*2/*3, upto(4), downto(1), shortname( #format $val(me::levelname) ),
        levelnames(6 or less, 9 or less, 12 or less, 15 or less), group(Chronic Pain), page(B126),
        displaynameformula( $val(me::name): Attack occurs on a roll of $val(me::levelname) )
```

#GroupList

From GCA wiki, by Armin; modified.

The #GROUPLIST directive allows you to include a comma separated list of items based on an existing Group. By default, the name of the items obtained from the group will include the prefix tag.

> #GROUPLIST(GROUPNAME [, APPEND(TEXT), PREPEND(TEXT), FLAGS(FLAGLIST)])

GROUPNAME is the name of the Group to be used, as defined in a Group section of a data file. It may be enclosed in quotes (must be if the name includes a comma) and may include the GR: prefix tag.

All of these tags are optional.

APPEND(TEXT) allows you to include a bit of text that will be added to the end of each item from the group. Do not enclose the append text in quotes, although you may use quotes as part of the append text if you wish.

PREPEND(TEXT) allows you to include a bit of text that will be added to the front of each item from the group. Do not enclose the prepend text in quotes, although you may use quotes as part of the prepend text if you wish.

FLAGS(FLAGLIST) allows you to include flags that alter the behavior of the list generation. One or more flags may be included in the FLAGLIST, and if more than one, they should be separated by commas.

The flags that may be used in the FLAGLIST are:

NOPREFIX to prevent the inclusion of the prefix tags in the names of group items.

INQUOTES to enclose each list item within quotes. Any appended or prepended text will appear outside the quotes.

INPARENS to enclose each list item within parentheses. Any appended or prepended text will appear outside the parens.

INBRACES to enclose each list item within curly {} braces. Any appended or prepended text will appear outside the braces.

MASTERQUOTES to have GCA place quotes around the whole item and appended/prepended text combination.

MASTERBRACES to have GCA place curly {} braces around the whole item and appended/prepended text combination.

Note: #GROUPLIST is an older directive, and the APPEND(), PREPEND(), and FLAGS() tags are replaced by the TEMPLATE([SOME TEXT] %LISTITEM% [MORE TEXT]) tag in newer directives. #GROUPLIST may eventually be updated to use the TEMPLATE() tag as well, but if so, the older tags will likely continue to be supported for backward compatibility.

#Input

The #Input directive allows you to get a simple bit of text from the user.

> #INPUT(["PROMPT" [, "DEFAULT TEXT" [, "TITLE"]]])

PROMPT	is the text of the prompt displayed to the user along with the input box. This value should be enclosed in quotes or braces if necessary.
DEFAULT TEXT	is the default value of the text to be input by the user. This value should be enclosed in quotes or braces if necessary. This may be changed by the user.
TITLE	is the title of the input window. This value should be enclosed in quotes or braces if necessary.

This directive is especially useful for having users enter a particular specialty or other similar use, when there isn't a pre-defined list. If the user presses the Cancel button, or enters no text, the item won't be added to the character.

Result Variables

Result variables are replaced by the selection made by the user. Result variables may appear anywhere in the trait definition.

%INPUT%	GCA will insert the user's entered text in place of this variable.

Note: As with other directives, GCA will remove this directive from the trait when it is added to the character. Unlike with other directives, the output generated by this directive will be inserted in place of the result variables, instead of in place of this directive.

#InputReplace

This directive is like the #INPUT directive, in that it gets input from the user, but it also allows for a custom bit of text that the routine will look for and replace with the input from the user.

> #INPUTREPLACE([PROMPT,] TARGETTEXT)

or

> #INPUTREPLACE(PROMPT, TARGETTEXT [, DEFAULT TEXT [, TITLE]])

PROMPT	is the text of the prompt displayed to the user along with the input box. This value should be enclosed in quotes or braces if necessary.
TARGETTEXT	is text that should be replaced by the text entered by the user. This value should be enclosed in quotes or braces if necessary.
	Note that to be effective, the TARGETTEXT should appear in the tag list somewhere, and it should be unique.

DEFAULT TEXT	is the default value of the text to be input by the user. This value should be enclosed in quotes or braces if necessary. This may be changed by the user.
TITLE	is the title of the input window. This value should be enclosed in quotes or braces if necessary.

#InputToTag

This directive is very much like the #INPUTREPLACE directive above, except that the text from the user is being inserted into a specific tag on the trait.

```
#INPUTTOTAG( PROMPT, TARGETTAG [, DEFAULT TEXT [, TITLE ]] )
```

PROMPT	is the text of the prompt displayed to the user along with the input box. This value should be enclosed in quotes or braces if necessary.
TARGETTAG	is the tag whose value should become the text entered by the user. This value should be enclosed in quotes or braces if necessary.
DEFAULT TEXT	is the default value of the text to be input by the user. This value should be enclosed in quotes or braces if necessary. This may be changed by the user.
TITLE	is the title of the input window. This value should be enclosed in quotes or braces if necessary.

If the TARGETTAG already has a value, #INPUTTOTAG will be skipped, and no value will be requested from the user.

If you include the TITLE, you must also include the DEFAULT TEXT, although it can simply be left empty, like this:

```
#InputToTag( "Enter a name extension:", nameext, , "Input Name Extension" )
```

#InputToTagReplace

This directive is the same as the #INPUTTOTAG directive above, except that the text from the user will replace any existing value of the specified tag.

```
#INPUTTOTAGREPLACE( PROMPT, TARGETTAG [, DEFAULT TEXT [, TITLE ]] )
```

PROMPT	is the text of the prompt displayed to the user along with the input box. This value should be enclosed in quotes or braces if necessary.
TARGETTAG	is the tag whose value should become the text entered by the user. This value should be enclosed in quotes or braces if necessary.
DEFAULT TEXT	is the default value of the text to be input by the user. This value should be enclosed in quotes or braces if necessary. This may be changed by the user.
TITLE	is the title of the input window. This value should be enclosed in quotes or braces if necessary.

If the TARGETTAG already has a value, #INPUTTOTAGREPLACE will replace the value with the text entered by the user.

If you include the TITLE, you must also include the DEFAULT TEXT, although it can simply be left empty, like this:

```
#InputToTag( "Enter a name extension:", nameext, , "Input Name Extension" )
```

#List

From GCA wiki, by Armin; modified.

The #LIST directive allows you to include a comma separated list of items based on an existing List.

```
#LIST(LISTNAME [, APPEND(TEXT), PREPEND(TEXT), FLAGS(FLAGLIST) ] )
```

LISTNAME	is the name of the List to be used, as defined in a List section of a data file. It may be enclosed in quotes (must be if the name includes a comma) and may have the LI: prefix tag.

All of these tags are optional.

APPEND(TEXT)	allows you to include a bit of text that will be added to the end of each item from the group. Do not enclose the append text in quotes, although you may use quotes as part of the append text if you wish.
PREPEND(TEXT)	allows you to include a bit of text that will be added to the front of each item from the group. Do not enclose the prepend text in quotes, although you may use quotes as part of the prepend text if you wish.
FLAGS(FLAGLIST)	allows you to include flags that alter the behavior of the list generation. One or more flags may be included in the FLAGLIST, and if more than one, they should be separated by commas.

The flags that may be used in the FLAGLIST are:

INQUOTES to enclose each list item within quotes. Any appended or prepended text will appear outside the quotes.

INPARENS to enclose each list item within parentheses. Any appended or prepended text will appear outside the parens.

INBRACES to enclose each list item within curly {} braces. Any appended or prepended text will appear outside the braces.

MASTERQUOTES to have GCA place quotes around the whole item and appended/prepended text combination.

MasterBraces to have GCA place curly {} braces around the whole item and appended/prepended text combination.

Note: #List is an older directive, and the append(), prepend(), and flags() tags are replaced by the template([some text] %ListItem% [more text]) tag in newer directives. #List may eventually be updated to use the template() tag as well, but if so, the older tags will likely continue to be supported for backward compatibility.

#Message

The #Message directive allows you to display a message to the user when the trait is added to the character.

```
#Message(Message Text)
```

Message Text	is the text to be displayed to the user in a message box.

#Ref

Provides a way to reference modifiers, rather than defining them, in various functional tags.

Supported in InitMods(), AddMods(), Adds(), Creates().

I'm going to use these two modifiers in examples below:

```
[MODIFIERS]
<Self-Control>
Self-Control Roll, *0.5/*1/*1.5/*2, upto(4), downto(1), group(Self-Control), page(B121),
        shortname(#format $val(me::shortlevelname) ),
        levelnames("15 or less, almost all the time",
                   "12 or less, quite often",
                   "9 or less, fairly often",
                   "6 or less, quite rarely"),
        shortlevelnames(15,
                   12,
                   9,
                   6),
        displaynameformula( You resist on a roll of $val(me::levelname) )

<Chronic Pain>
Interval, *0.5/*1/*1.5/*2, upto(4), downto(1), shortname( #format $val(me::levelname) ),
        levelnames(1 hour, 2 hours, 4 hours, 8 hours), group(Chronic Pain), page(B126),
        displaynameformula( $val(me::name): $val(me::levelname) )

Frequency, *0.5/*1/*2/*3, upto(4), downto(1), shortname( #format $val(me::levelname) ),
        levelnames(6 or less, 9 or less, 12 or less, 15 or less), group(Chronic Pain), page(B126),
        displaynameformula( $val(me::name): Attack occurs on a roll of $val(me::levelname) )
```

InitMods()

#REF allows for specifying modifiers by reference in INITMODS(), and other places, so that an entire modifier definition no longer needs to be included. The existing behavior is preserved, but now you can also specify a reference to a modifier instead, using the #REF directive and this format:

```
INITMODS( #REF MODIFIERNAME [ = X ]  [ FROM GROUPNAME ] )
```

Braces are optional if not needed to protect parsing on the = or FROM keywords, or the pipes separating various INITMODS() items. (If the modifier has a name extension, you do need to include it inside parens as part of MODIFIERNAME, as per usual.)

The assignment is optional, but if used sets the initial level of the modifier.

The FROM GROUPNAME section is optional if the modifier is also found in one of the defined MODS() modifier groups, but required if not.

Do NOT include the #REF, FROM, or assignment within any braces around names. (You may include the entire statement inside quotes or braces to separate it from other modifier blocks within the INITMODS().)

Using the Self-Control Roll modifier example from the example modifiers included above, you can refer to it in an INITMODS() like this:

```
initmods(#ref Self-Control Roll = 2 FROM Self-Control)
```

which will look for it in the Modifiers group called Self-Control and assign it at level 2 (12 or less).

In a trait, it might look like this:

```
[DISADVANTAGES]
<Mundane Mental>
Chronic Depression, -15, mods(Self-Control), page(B126), cat(Mundane, Mental), initmods( #ref Self-Control
Roll = 2 )
```

Note that because the mods(Self-Control) tag exists in this item, GCA will find the modifier even though we don't explicitly provide a group to look in.

You can mix with the full-definition method, too:

```
[DISADVANTAGES]
<Mundane Mental>
Chronic Depression, -15, mods(Self-Control), page(B126), cat(Mundane, Mental),
initmods( #ref Self-Control Roll = 2 | _
        Mitigator: Meds, -60%, group(_General), page(B112), mitigator(yes), shortname(w/Meds)_
        )
```

AddMods()

#REF is also supported in ADDMODS(). The existing behavior is preserved, but now you can also use this new format:

```
ADDMODS( #REF MODIFIERNAME [ = X ]  [ FROM GROUPNAME ] TO TARGETTRAIT )
```

This can also be mixed with the other valid ADDMODS() references (MODGROUP:MODNAME or #NEWMOD()). For example, using the Chronic Pain modifier group example modifiers included above:

```
addmods( ( "Chronic Pain:Interval", "#ref Frequency=2 from Chronic Pain" ) to "DI:Chronic Pain" )
```

The first reference is the existing format of MODGROUP:MODNAME, and the second is the new format using #REF. Note also that the new reference format allows for specifying a level for leveled modifiers, while the old reference format does not.

Keep in mind that if you leave off the FROM portion in this usage, the MODS() tag of the TARGETTRAIT will be used to find the modifier. In our example above, we could have used

```
#ref Frequency=2
```

because the Chronic Pain disadvantage has the Chronic Pain modifier group in its MODS() tag.

Adds()

#REF is also supported in the WITH and AND blocks of the ADDS() tag, instead of having to include full definitions.

The existing behavior is preserved, but now you can also use this new format to reference existing modifiers:

```
ADDS( TRAIT _
        [ WITH "#REF MODIFIERNAME [ = X] [ FROM GROUPNAME ]" _ ]
        [ AND  "#REF MODIFIERNAME [ = X] [ FROM GROUPNAME ]" _ ]
)
```

For example:

```
Chronic Depression 4, -15, mods(Self-Control), page(B126), cat(Mundane, Mental),
        adds(_
                DI:Chronic Pain=3 _
                        with " #ref Interval=3 " _
                        and  #ref Frequency _
                )
```

If the FROM block is left off, the MODS() tag of the newly added trait will be used when searching for the specified modifiers. In the example above, Interval and Frequency will be found because the Chronic Pain disadvantage has those in its MODS() tag.

Creates()

#REF is also supported in the WITH and AND blocks of the CREATES() tag, instead of having to include full definitions. See ADDS() above for details.

DATA FILE COMMANDS

This section will cover what you need to know to use data file commands in GDFs.

Commands are used in data files and are placed one command per line. They may come in any section (except Author), but the sections aren't applicable to the commands (that is, the section in which you place the command has no effect on how the command works, or what the command does). Commands only affect data that has already been loaded. No commands can affect data that has not yet been loaded.

#AddToList

#ADDTOLIST allows you to add items to an existing LIST defined elsewhere or create the list and add items to it.

```
#ADDTOLIST "LI:LISTNAME" ITEMSTOADD
```

LI:LISTNAME	The LI: prefix tag is optional. Quotes (") or braces ({ }) are encouraged around the list name, but they are optional unless it includes a space.
ITEMSTOADD	The comma separated list of entries to add to the list. Quotes (") or braces ({ }) are optional around each item being added unless it includes a comma.

```
#AddToList "LI:Barney" Whatever, "Whatever (Two)", More Stuff
#AddToList Barney Whatever, Whatever (Two), More Stuff
```

In the example lines shown, the three items Whatever, Whatever (Two), and More Stuff would each be added to the list Barney.

#AddToListByTag

Creates a new list, or adds to an existing one, and adds all library traits that match the specified selection criteria. Allows for an optional template for output.

```
#ADDTOLISTBYTAG "LI:LISTNAME" [ TEMPLATE(TEXT) ] SELECTITEMSBYTAG
```

"LI:LISTNAME"	The name of the list that you're targeting. The LI: is optional, as are enclosing quotes or braces unless the list name includes a space. Follow with a space to separate it from the TEMPLATE() tag or the selection criteria.
TEMPLATE(TEXT)	The TEMPLATE() tag is optional. This should be separated from the list name and the selection criteria by spaces. Since this option is in tag format, the parens should protect it from parsing issues, but you do have to ensure that you don't mismatch quotes, parens, or braces.
SELECTITEMSBYTAG	is the comma separated list of selection criteria, using the SelectItemsByTag format.

The TEMPLATE() tag makes use of some templating support that I wrote to allow for customizing the Info display in the GUI. This means that it knows what trait it's looking at, and just needs to potentially fill in values. You can reference any tags from the current trait by enclosing them in % characters, such as %nameext% for name extension, or %points% for points. There are a few special case variables that require extra processing and are therefore enclosed in double %% signs, such as %%COSTPROG%%, but most of those apply only to character traits, not system traits, so aren't applicable here.

An example template might be template(%name%, %type%) for a skill, which would become Accounting, IQ/H for the Accounting skill.

As an experimental addition, I've hooked up the TextFunctionSolver, so you have some additional power available. This is an experiment because most of the Solver features were designed to operate on character traits, so some things just won't work correctly. I've added some traps to allow for functions that don't depend on character traits to work and not cause problems, so I think we're good there, but actively using such a feature will still probably crash something.

The TextFunctionSolver will normally run after the template variables are replaced by the trait tag values. However, if the very first piece of your TEMPLATE() text is $FIRST, then GCA will remove that flag and call the TextFunctionSolver first, then replace variables, and then do the normal pass through TextFunctionSolver if it looks like there might still be text functions in there.

An example using text functions might look like this:

```
template($if(@len(%nameext%)=0 then "%name%" else "%name% (%nameext%)"))
```

That is rather pointless as it's replicating just using %fullname%, but it's a functional example.

If TEMPLATE() is left out, what is added to the list is the selected trait's FULLNAME property.

Example:

```
#AddToListByTag "LI:TestingStuffList" Advantages, Text, name isnotempty echolog
```

This command will select every Advantage and add the full name for each one to the TestingStuffList, because every valid Advantage has a name that is not empty. (It will also spit some processing info into the log as it works.)

#Clone

From GCA wiki, by Foxfire; modified.

This command allows for duplicating an existing trait as a new trait.

```
#CLONE "TRAITNAME" AS "NEWTRAITNAME"
```

TRAITNAME	is the name of the existing trait to be duplicated. The full name, including prefix tag, should be specified.
NEWTRAITNAME	is the new name to give the duplicated trait. The full name, including prefix tag, should be specified.

This can be useful to create modified traits using other commands without changing the original. This command can also be used to rename a trait, in combination with the #Delete command.

Example:

> #Clone "AD:Magery" as "AD:Spirit Magery"

creates an advantage called Spirit Magery which has all of the same effects, requirements, etc. as Magery.

#CloneMod

This works like #CLONE but clones a modifier.

> #CLONEMOD SOURCEGROUP:SOURCEMODIFIER AS NEWGROUP:NEWNAME

Unlike traits, modifiers don't have an item type, but they're unique within their groups, so #CLONEMOD requires specifying the modifier group and the full name of the modifier, as shown.

Enclose the entire SOURCE or NEW sections in quotes or braces as required.

#Delete

From GCA wiki, by Foxfire; modified.

This command allows for deleting a trait from the loaded data.

> #DELETE "TRAITNAME"

TRAITNAME	is the trait to be deleted. The full name, including prefix tag, should be specified.

Note that deleting traits may have strange effects if there are traits which require the deleted trait as a prerequisite.

Example:

> #Delete "AD:Medium"

Removes the advantage Medium from the list of advantages.

#DeleteAllCats

Instructs GCA to delete all the categories created up to this point for a particular type of trait.

> #DELETEALLCATS ITEMTYPE

ITEMTYPE must be one of the standard types GCA understands, or ALL to delete all categories for all trait types.

#DeleteByTag

From GCA wiki, by Bjork; modified.

This command allows for removing traits from loaded data based on the value of a specified tag.

```
#DeleteByTag TraitList, Commands, Tag=Value [ unless Tag=Value ]
```

TraitList	is the list to look in, such as Skills, Equipment, or All for everything.
Commands	is everything needed to specify what GCA should do with the comparison; usually, this means specifying Num or # for a numeric comparison, or $ or Text for a text comparison; this will vary by tag. You can also specify IgnoreEmpty to have GCA ignore tags that are empty (this prevents IsEmpty or IsNotEmpty from working; see below), and EchoLog if you want to fill your log with endless lines of GCA showing you what it looked at and did or didn't delete as a result. (You may use EchoLogShort instead; EchoLogShort will only print the command itself, and each trait that's being deleted, to the log.)
Tag=Value	is the name of the tag, the comparison to make, and the value you're looking for. The tag name should be exactly as used by GCA, and the value should be the exact text or number to compare against. Valid comparisons are >, <, =, >=, <=, or <>. Also, because comparing text for/against an empty tag can result in unexpected behavior during text comparisons (numeric comparisons convert empty tags to 0, so use IgnoreEmpty for those if 0 doesn't work for you in such cases), there are two special text comparison operators that have no =Value portion, and these are IsEmpty and IsNotEmpty. IsEmpty counts as True if the specified tag is empty, and IsNotEmpty counts as True if the tag is not empty.
unless Tag=Value	is an exception block, to allow for an exception to the rule. Unless supports all the same comparisons as the standard clause. You create an unless section by including the keyword unless followed by the comparison to be made. You may also include the same Numeric or Text designations immediately after the unless keyword if you want to use a different type from the base comparison.

Here are some examples:

```
#DeleteByTag Equipment, Num IgnoreEmpty, techlvl < 5
```

This command will delete any equipment items that have specified a techlvl() tag, and for which the numeric value contained in that tag is less than 5.

Note, however, that Numeric comparisons convert text values to numbers, and if there's not a number as the text value, it is probably going to convert to 0. This means that in our example here, traits such as _New Armor, that have a techlvl([techlevel]) like this, will appear to evaluate to techlvl(0), which means in our example it would be deleted. To avoid this kind of result, you'll probably need to do a Text comparison, and compare each techlvl() you want to delete, like so:

```
#DeleteByTag Equipment, Text IgnoreEmpty, techlvl = 0
#DeleteByTag Equipment, Text IgnoreEmpty, techlvl = 1
#DeleteByTag Equipment, Text IgnoreEmpty, techlvl = 2
#DeleteByTag Equipment, Text IgnoreEmpty, techlvl = 3
#DeleteByTag Equipment, Text IgnoreEmpty, techlvl = 4
```

Be aware that in this particular example, there are a number of traits, such as _New Innate Attack, that are currently marked as techlvl(0), which you probably don't really want to delete. But, for our purposes here, this set of commands would delete everything below techlvl(5), but not delete traits with other text values.

Here is an example using UNLESS:

```
#DeleteByTag Equipment, Numeric IgnoreEmpty EchoLog, techlvl >= 5 unless $ techlvl contains ^
```

This example will delete any equipment item that has a non-empty techlvl() tag that equates to a value greater than or equal to 5, unless it contains the text ^, in which case it isn't deleted.

And here's a simple example covering all traits:

```
#DeleteByTag All, Text, x IsNotEmpty
```

This command will delete any trait where the x() tag has anything in it at all. (This would remove pretty much everything in GCA that appears with a little wrench icon next to it in the Available lists.)

Text Comparison Special Case

TAG=VALUE can use the format TAG HASINLIST VALUE or TAG CONTAINS VALUE. In the HASINLIST case, the TAG value being checked will be considered to be a comma delimited list, and the VALUE must match one of the items in that list. In the Contains case, the whole string of text in the TAG will be looked at, and if VALUE appears anywhere in the string, it will be considered to match. Both of these are case insensitive.

TAG=VALUE can also use the format TAG ISNUMERIC or TAG ISNOTNUMERIC. These test if the TAG value would qualify as a numeric value or not. While making a numeric test, remember that non-numeric values usually evaluate to 0 (or the value of any numbers beginning the text). If testing ISNUMERIC, it only qualifies as true if the entire value qualifies as a number. An example:

```
#DeleteByTag Equipment, Numeric IgnoreEmpty EchoLog, techlvl < 5 unless $ techlvl IsNotNumeric
```

A numeric test of the techlvl() tag would be done, and a number of special cases would be deleted if not saved by the UNLESS clause, which allows for preserving those values due to the fact that they're not actually numbers, just text that were converted to 0 for the simple numeric comparison.

#DeleteCat

Instructs GCA to delete the specified category for a particular type of trait.

```
#DELETEALLCATS ITEMTYPE, CATEGORY
```

ITEMTYPE	must be one of the standard types GCA understands, or ALL to delete the category from all types of traits.
CATEGORY	is the name of the category to be deleted. Can be enclosed in quotes or braces.

#DeleteFromGroup

This command allows for removing Group items from a group.

```
#DELETEFROMGROUP GROUPNAME TRAIT [, TRAIT2 ][, ALL TRAIT3 ]
```

The group name should be the first thing after the command, separated by spaces from the command and the list of items that follows. Then should be a list of items, separated by commas, that should be deleted from the specified group. You can use the ALL keyword before an item to specify that all such items should be deleted—this allows you to skip listing every possible name extension for specialized skills that may be in the group. Quotes should be used around any name that includes a comma, or around the group name if it includes a space. The GR: prefix may be used but is not required for the group name.

#DeleteFromList

Allows you to delete entries from a previously created LIST.

```
#DELETEFROMLIST "LI:LISTNAME" ITEMSTODELETE
```

#DELETEFROMLIST uses the same format as #ADDTOLIST, just the command name differs.

LI:LISTNAME	The LI: prefix tag is optional. Quotes (") or braces ({ }) are encouraged around the list name, but they are optional unless it includes a space.
ITEMSTODELETE	The comma separated list of entries to delete from the list. Quotes (") or braces ({ }) are optional around each item unless it includes a comma.

```
#DeleteFromList "LI:Barney" Whatever, "Whatever (Two)", More Stuff
#DeleteFromList Barney Whatever, Whatever (Two), More Stuff
```

In the example lines shown, the three items Whatever, Whatever (Two), and More Stuff would each be removed from the list Barney.

#DeleteModGroup

Allows for deleting an entire modifier group and all its modifiers at once.

```
#DELETEMODGROUP "TARGETGROUP"
```

Specify the mod group name to remove, and it is removed, along with all the modifiers it contains.

For example:

```
#DeleteModGroup "Burning Attack Limitations"
```

#DeleteModsFromGroup

Allows for deleting specific modifiers from existing data.

```
#DELETEMODSFROMGROUP "TARGETGROUP" MODNAME [, MODNAME2 [ ... ] ]
```

Similar to #DELETEFROMGROUP in structure, you specify the modifier group from which you're deleting items, then a space, then all the modifiers from that group that you want to delete, in a comma separated list. Enclose the modifier group name in quotes and enclose any modifier names that contain commas inside quotes as well.

For example:

```
#DeleteModsFromGroup "Burning Attack Enhancements" Partial Dice, "Partial Dice, Per Die"
```

#Get

#GET allows you to retrieve #STORE values.

#GET, and its companion #STORE, are for use in building other library features, such as SELECT() or #CHOICE dialogs. You can use them to insert what will be character features, such as storing a formula that's inserted into a trait. Remember that #GET is only processed when a trait is added to the character.

The #GET template looks like this:

```
#GET( VARNAME )
```

and simply replaces the entirety of the #GET command with the retrieved value.

A couple examples:

```
[ADS]
<_TESTING>
#Store TestData=This is a Test
Example 1 (#GET(TESTDATA)), 5/10

#Store DialogTitle=Test Dialog
#Store DialogText = Hello!~PThis is a dialog to demonstrate the #Store and #Get commands.~PTry it!
#Store DialogList = Item 1 Chosen, Item 2 Chosen, Item 3 Chosen
Example 2 (%choice%), 5/10, x(#ChoiceList(list(#get(dialoglist)),
        Title(#get(dialogtitle)),Text(#get(dialogtext)) ))
```

In Example 1, the name extension becomes This is a Test when added to the character.

In Example 2, a CHOICE dialog pops up when added to a character, and the dialog is populated with the #STORE values.

#IF

Allows you to have conditional file processing.

The structure is like this:

```
#IF WANT = VALUE [THEN]
        [...]
[#ELSEIF WANT2 = VALUE2 [THEN]]
        [...]
[#ELSE]
```

```
    [...]
#END[IF]
```

Notice that the THEN keywords are optional; GCA will ignore them if found, but you can simply leave them out if desired. Same with the IF in #ENDIF; GCA considers any #END to be the end of the current #IF structure.

You may nest #IF..#END blocks.

Note that there is *no* expression evaluator involved here. Support exists for a tiny set of very specific comparisons, which I'll cover here. If you try anything else, GCA will consider the block FALSE and continue on, happily ignoring that section (and provide an error in the log if you have verbose book processing turned on).

The two types of WANT = VALUE comparisons currently supported are these:

1) Check for a file

```
FILELOADED = NAMEOFFILE
```

This allows you to see if a file has been loaded before this one. The file currently being processed does *not* count. You must use the exact file name that GCA has loaded (ignoring path information and capitalization).

GCA supports the following aliases for FILELOADED, so you may use whichever you remember: FILELOADED, LOADEDFILE, FILEISLOADED, LOADED, BOOKLOADED, BOOKISLOADED.

NAMEOFFILE may be in quotes or braces.

2) Check for a trait

```
TRAITLOADED = NAMEOFTRAIT
```

This allows you to see if a particular trait exists in the current library data. Anything loaded before this comparison is a possibly valid subject.

GCA supports the following aliases for TRAITLOADED, so you may use whichever you remember: TRAITLOADED, TRAITEXISTS, TRAITPRESENT, LOADEDTRAIT.

NAMEOFTRAIT may be in quotes or braces, and must be in the standard fully qualified format, with prefix and full name and extension, as applicable.

#Log

Allows the author to send a message to the Logging pane.

```
#LOG [COLOR] MESSAGE
```

MESSAGE can be enclosed in quotes and COLOR is optional from RED, BLUE, GREEN, or ORANGE (not specified is black).

#MergeModTags

Works like #MERGETAGS, but for modifiers.

```
#MERGEMODTAGS IN TARGETGROUP:TARGETMODIFIER WITH TAGLIST
```

Unlike traits, modifiers don't have an item type, but they're unique within their groups, so #MERGEMODTAGS requires specifying the modifier group and the full name of the modifier, as shown.

For example:

```
#mergemodtags in "Burning Attack Enhancements:Partial Dice" with "page(EX)"
```

The ALL keyword is **not** supported.

#MergeTags

From GCA wiki, by Foxfire; modified.

This command can be used to add tags to an existing trait, or add to tags which already exist for a trait.

```
#MERGETAGS IN [ALL] "TRAIT" WITH TAG[, TAG]
```

ALL	is optional; if included all traits which have a base name of TRAIT will have the new tags added.
TRAIT	is the base name of the trait to modify. Prefix tags (such as AD: or DI:) should be used. Unless using the ALL option, this should be the full name of the trait as defined in the original data file, including the name extension if present. The trait name should be enclosed in quotes, although this is technically optional if the trait name does not contain any spaces.
TAG	is the new tag and tag value to add to the trait, in the form of TAGNAME(TAGINFO). Note that multiple tags may be merged by using a comma-delimited list of tag information.

Examples:

```
#MergeTags in all "AD:Jumper" with taboo(AD:Magery 0)
```

Note that the GCA file for Basic Set: Characters defines two Jumper advantages: Jumper (World) and Jumper (Time). The above command would make both of these advantages incompatible with having the advantage Magery 0. If you also had the Powers file loaded already, the advantage Jumper (Spirit) would also be modified in the same way.

```
#MergeTags in "AD:Jumper (Time)" with needs(AD:Magery 0)
```

This line would make Jumper (Time) require the advantage Magery 0 but would not change Jumper (World).

```
#MergeTags in "AD:Magery" with needs(AD:Medium)
```

As listed in the Basic Set data file, Magery already has the tag needs(AD:Magery 0). The above command would force Magery to also require the Medium advantage, making the tag needs(AD:Magery 0, AD:Medium).

#ModifierGroupWarnings

This has only two options, On or Off, like so:

```
#ModifierGroupWarnings On
#ModifierGroupWarnings Off
```

This is OFF by default. When ON (and #VERBOSE is also ON), this command will cause GCA to spit out to the log every time it replaces the modifier's included GROUP() with the one specified by the sub-head group in the file. The group specified in the file must be the official group source for the data and is considered part of the name of the modifier. However, there are reasons that having a GROUP() specified in the tag block can be convenient.

The default behavior of GCA's book processing has now changed from what was the equivalent of this command being ON to this command being OFF.

#ReplaceTags

From GCA wiki, by Foxfire; modified.

This command replaces existing tags for a trait.

```
#REPLACETAGS IN [ALL] TRAIT with TAG [, TAG ]
```

ALL	is optional; if included, all traits which have a base name of TRAIT will have the tags replaced.
TRAIT	is the base name of the trait to modify. Prefix tags (such as AD: or DI:) should be used. Unless using the ALL option, this should be the full name of the trait as defined in the original data file, including the name extension if present. The trait name should be enclosed in quotes, although this is technically optional if the trait name does not contain any spaces.
WITH	required keyword that separates the trait being operated on from the tag(s) being applied.
TAG	is the new tag to add to the trait, in the form of TAGNAME(TAGINFO). Note that multiple tags may be replaced by using a comma-delimited list of tag information.

Example:

```
#ReplaceTags in "AD:Magery" with needs(AD:Medium)
```

As listed in the GCA file Basic Set: Characters, Magery has the tag needs(AD:Magery 0). The above command would replace this, effectively making Magery require the Medium advantage instead of Magery 0.

#ReplaceModTags

Works like #REPLACETAGS, but for modifiers.

```
#REPLACEMODTAGS IN TARGETGROUP:TARGETMODIFIER WITH TAGLIST
```

Unlike traits, modifiers don't have an item type, but they're unique within their groups, so #REPLACEMODTAGS requires specifying the modifier group and the full name of the modifier, as shown.

For example:

```
#replacemodtags in "Burning Attack Enhancements:Partial Dice" with "cost(+1/+2),formula(%level * 5)"
```

The ALL keyword is **not** supported.

#Store

#STORE is a data file command that allows storing text as a named block for use elsewhere in the Library, such as

```
#store warning=Don't Do That!
```

These are basically global variables on the Library level, rather than at the trait level, and in concept are similar to Lists, but store only a single item of text, rather than a group of related text items.

#STORE, and its companion #GET, are for use in building other library features, such as SELECT() or #CHOICE dialogs. You can use them to insert what will be character features, such as storing a formula that's inserted into a trait. Remember that #GET is only processed when a trait is added to the character.

The #STORE template looks like this:

```
#STORE VARNAME = TEXT
```

The VARNAME should be simple, but if necessary (when it includes an = sign) it can be enclosed in quotes or braces.

The TEXT bit can be whatever text you want, so long as it conforms to GCA's data file rules (one line, or put together from multiple lines using line continuation). Don't enclose TEXT in quotes or braces unless you want them wherever the text is going to be inserted. Includes support for ~P (that's a tilde followed by a capital P) for inserting a carriage return/line feed into the text, which is converted during #GET retrieval.

Be conscious of where the text is going to go; don't include things like unbalanced parens or other characters that might break the parsing of the destination after the value is inserted.

Whitespace at the front or end of either part is trimmed.

See #GET for examples of use.

#Verbose

This has only two options, On or Off, like so:

```
#Verbose On
#Verbose Off
```

This allows you to turn on VerboseBookProcessing for a specific file, or portion of a file, for testing purposes, without having to have full-on verbosity for every file you're loading. (Technically, this sets a different property, and does not affect the user's VerboseBookProcessing setting at all; if they have that turned on, this will have no impact on it at all, and they'll get the verbosity they desire.)

It should be considered polite to remove these directives from files before they're made publicly available, as many users will find verbose book processing quite annoying.

SPECIAL NOTES

Gains Bonuses

Gains bonuses are bonuses that a trait claims for itself, based on some other trait, rather than having to edit that other trait to have it give a bonus. GCA has never supported Gains bonuses, instead requiring that you create a work-around using Gives bonuses, when a Gains would have been more convenient.

Now, we have initial support for Gains bonuses, using the existing structure and the existing GIVES() tag, but with a tiny change in wording, using a FROM keyword instead of the TO keyword.

See the GIVES() tag for more information.

Bonus Targets

When granting bonuses, knowing the right target is important, and the right target isn't always a trait. To grant bonuses to items that aren't specifically individual traits, here are the targets you can use.

By Prefix

When you want to grant a bonus to a target based on its category (specific by trait type), use one of the prefixes given here.

CA:	to a category of Advantages
ADCAT:	to a category of Advantages
DICAT:	to a category of Disadvantages
DISADCAT:	to a category of Disadvantages
LANGCAT:	to a category of Languages
LACAT:	to a category of Languages
CULTCAT:	to a category of Cultural Familiarities
CUCAT:	to a category of Cultural Familiarities
FEATURECAT:	to a category of Features
FECAT:	to a category of Features
PECAT:	to a category of Perks
PERKCAT:	to a category of Perks
EQCAT:	to a category of Equipment
EQUIPCAT:	to a category of Equipment
SKCAT:	to a category of Skills
SKILLCAT:	to a category of Skills
CL:	to a category of Skills (a Class of Skills)
SPCAT:	to a category of Spells
SPELLCAT:	to a category of Spells
CO:	to a category of Spells (a College of Spells)
GR:	to a Group (defined data file Group, not a modifier group)

By Keyword

When you want to target a bonus to specific things based on some pre-defined targets.

SKILLS	to all Skills (levels)
SKILLPOINTS	to all Skills (points)
SPELLS	to all Spells (levels)
SPELLPOINTS	to all Spells (points)
CULTURES	to all Cultural Familiarities
CULTURE	to all Cultural Familiarities
LANGUAGES	to all Languages
LANGUAGE	to all Languages
FEATURES	to all Features
FEATURE	to all Features
STSKILLS	to all Skills based on ST (type ST/?; levels)
DXSKILLS	to all Skills based on DX (type DX/?; levels)
IQSKILLS	to all Skills based on IQ (type IQ/?; levels)
HTSKILLS	to all Skills based on HT (type HT/?; levels)
STSKILLPOINTS	to all Skills based on ST (type ST/?; points)
DXSKILLPOINTS	to all Skills based on DX (type DX/?; points)
IQSKILLPOINTS	to all Skills based on IQ (type IQ/?; points)
HTSKILLPOINTS	to all Skills based on HT (type HT/?; points)
STSPELLS	to all Spells based on ST (type ST/?; levels)
DXSPELLS	to all Spells based on DX (type DX/?; levels)
IQSPELLS	to all Spells based on IQ (type IQ/?; levels)
HTSPELLS	to all Spells based on HT (type HT/?; levels)
STSPELLPOINTS	to all Spells based on ST (type ST/?; points)
DXSPELLPOINTS	to all Spells based on DX (type DX/?; points)
IQSPELLPOINTS	to all Spells based on IQ (type IQ/?; points)
HTSPELLPOINTS	to all Spells based on HT (type HT/?; points)

Parents and Children

Added support for PARENT:: as a target for bonuses and as a reference in the solver, so you can do something like having a scope as a piece of equipment that is added to a rifle as a child, and then grant a bonus to that parent rifle's acc. This will work for both the parent/child and the meta-trait/component relationships.

Added support for CHILD:: as a target for bonuses, so you can have a parent that grants the same bonus to anything that is made a child of it. This will work for both the parent/child and the meta-trait/component relationships. This is not available as a reference in the solver because there is no way to resolve what child of many may be intended.

#Any

This is a more specifically defined use of the Name Extension Directives detailed in Special Notes that I wanted to call out here.

You can now target a standard bonus (a target with a prefix tag and a name) to a trait name of #any, which means you can target a whole class of things for a specific targetable tag, if you wish. For example

```
-1 to EQ:#Any::minst
```

would allow you to reduce the MINST() requirements for all equipment items by 1 per level.

Modes

Modes have received more attention in GCA5 and have a new handler. To the user, this should be transparent, but it means that expecting mode-specific behavior from non-mode-specific tags may not work as expected.

Here are all the mode-specific tags:

Acc()
ArmorDivisor()
Break()
Bulk()
CharAcc()
CharArmorDivisor()
CharBlockScore()
CharBreak()
CharBulk()
CharDamage()
CharDamType()
CharEffectiveST()
CharMalf()
CharMinST()
CharParry()
CharParryScore()
CharRadius()
CharRangeHalfDam()
CharRangeMax()
CharRcl()
CharReach()
CharRof()
CharScopeAcc()
CharShots()
CharSkillScore()
CharSkillUsed()
CharSkillUsedKey()
Damage()
DamageBasedOn()
DamageIsText()
DamType()
Dmg()
ItemNotes()
LC()
Malf()
MinST()
MinSTBasedOn()
Mode()
Notes()
Parry()
Radius()
RangeHalfDam()
RangeMax()
Rcl()
Reach()
ReachBasedOn()
Rof()
ScopeAcc()
Shots()
SkillUsed()
STCap()
Uses()
Uses_Sections()
Uses_Settings()
Uses_Used()
VTTModeNotes()

Damage Mini-Modes

Damage mini-modes allow you to store more than one set of damage values in a single attack mode. If you apply them manually, you need to set DAMAGE(), DAMTYPE(), ARMORDIVISOR(), and RADIUS() manually, using comma-separated values, one per mini-mode desired. DAMAGE() is always required, but the other tags are not. You can also apply a mini-mode via a bonus to MINIMODE$ in standard damage notation, such as 1d cut or sw+1 (2) burn (3); GCA will parse those damage strings into the appropriate tags (remember to put armor divisor and radius in parens, and that armor divisor comes before the damage type while radius comes after it).

This is considered an experimental feature, but it is ON by default. It can be turned OFF on the Experimental Features panel on the Program Options tab of the Options dialog.

NOTE: If you have mini-modes that you created manually by editing tag values, turning the option OFF in Options may result in a crash as that can result in illegal data. However, if you apply mini-modes via a bonus, that bonus will simply no longer get processed, so no crash will result.

The new DamageDisplayText function of a Mode will return the damage string correctly, whether there are mini-modes in use or not. When mini-modes are in use, damages will be displayed as comma-separated damage notations.

Using mini-modes results in some new calculated tags being managed in each Mode: MINIMODE_DAMAGE(), MINIMODE_DAMTYPE(), MINIMODE_ARMORDIVISOR(), and MINIMODE_RADIUS(). These shadow the original tags but have all the extra mini-mode data included, or slots for them when values are missing or empty.

For plugin authors, if you need to access mode data for purposes other than display, you can use the new GetMiniModes() function of a Mode. This returns all the mini-modes as part of a ModeManager, as full modes using the data of the mode in question, but the damage data for each mini-mode. That is, if you have a mode with two mini-modes of '1d cut' and 'sw+1 (2) burn (3)', then GetMiniModes() will return two modes that are identical to the original mode, but with the DAMAGE(), DAMTYPE(), ARMORDIVISOR(), and RADIUS() tags set for each type of damage for each of the two mini-modes.

Substitutions

Support has been added for a Substitutions system, which allows one trait to substitute for another when GCA is tracking down references. This system applies to character traits and does not do substitutions on a library basis.

With this system, you could create a Guns (Small) skill that substitutes for Guns (Pistol), and then when a reference to Guns (Pistol) is being sought by GCA, Guns (Small) would be considered a valid return.

You create a substitution table by using the SUBSFOR() tag in the item that is intended to substitute for the others. For example, our Guns (Small) trait would include subsfor("SK:Guns (Pistol)").

Multiple entries can be submitted at once in SUBSFOR() by separating them with commas. The trait type prefix code is required in the SUBSFOR() definition.

Implementation is currently limited, but should be working in most places, such as defaults and weapons tables. Note that the item that is the substitute also qualifies as an existing instance of anything for which it substitutes, so adding one of those items after the substitution table is created would be considered a duplicate by GCA.

User Targetable Bonuses

Support has been added for "user targetable" bonuses, which will allow the user to choose the targets to which the bonus is meant to apply.

GCA will manage the new CHOSENTARGETS() tag, which will track the names of the target items, and provide a UI for it through the use of a new button on the Edit Traits dialog, which will pop up a pick list from which they can choose target items. When bonuses are generated by the trait, GCA will create one bonus for each target, in each case replacing the %CHOSENTARGET% with the name of the target item.

See the GIVES() tag for more information.

Name Extension Directives

GCA now supports some directives specific to name extensions, in specific circumstances.

#Any

#ANY will return the first valid non-zero trait with the given name.

> #ANY [OF LIST][EXCEPT LIST]

#Best

#BEST will return the best (highest non-zero).

> #BEST [OF LIST][EXCEPT LIST]

#Worst

#WORST will return the worst (lowest non-zero).

> #WORST [OF LIST][EXCEPT LIST]

#None

#NONE will look only at traits without a name extension.

> #NONE

Details

Each of these directives allows you to specify how you'd like to look for the trait, given possible variations of the name extension.

So, if you have several possible Flight advantages, but any one of them works just to say that you have Flight, you could use a reference like

> SK:Flight (#any)

to find any of them.

There are also optional clauses to contract the acceptable pool of traits by the name extensions found, should you wish to do that. All these directives, excluding #NONE, support the OF and EXCEPT clauses.

OF specifies the extensions that are valid, and only traits with one of those specified extensions will be considered.

EXCEPT specifies the extensions that are not valid, and only traits that don't have a specified extension will be considered.

LIST is a list of the applicable name extensions for that clause. Enclose individual list items within quotes or braces if they include a comma or one of the other keywords.

The OF and EXCEPT clauses should never have any reason to be used at the same time, but should you do so, order isn't important so long as the clauses always follow the #keyword.

#NONE doesn't support clauses because it retrieves only traits without extensions, so there's no extension to check against them.

Solver

The Solver respects these new directives, so they should be working almost any time GCA is looking up a trait.

Note that #BEST and #WORST can be tricky, and should be used cautiously, as results may not be as expected in all cases, depending on what tag or value you're looking for. Only numeric values can be judged.

The Text Function Solver also respects these directives. However, because it returns text values, there's no good way to determine best or worst, and it therefore treats all of them as #ANY, where any non-empty value satisfies the request.

Needs Checking

The Needs system also supports these directives when checking prerequisites for any traits that are properly identified with the correct prefix tags, and that have a valid base name.

GCA will assemble a list of all traits of the correct type that have the given base name, and then process them in this fashion:

#ANY - all traits will be evaluated and used to test the condition, but only until the first trait found that satisfies the given condition; or until all traits fail.

#BEST - all traits will be evaluated, and the one with the best (highest) value will be used to test the condition.

#WORST - all traits will be evaluated, and the one with the worst (lowest) value will be used to test the condition.

Obviously, this only works for numeric values and conditions that evaluate numeric expressions. Inactive traits are not valid and won't be considered.

Here's a completely arbitrary and silly example of the NEEDS() usage:

```
needs(SK:Animal Handling (#any of equines, big cats) = 12, SK:Animal Handling (#best) > 15, SK:Animal
Handling (#worst except "raptors") < 8)
```

In this example, only the Animal Handling (Equines) or Animal Handling (Big Cats) skills will satisfy the need for an Animal Handling skill at 12 or higher; but any version at all will work to satisfy the need that some skill is over 15; and finally, the worst Animal Handling skill must be less than 8 but can't be Animal Handling (Raptors) because that's excluded.

Bonus Targeting

The bonus targeting system supports some of these directives, as some don’t make sense in this context. A valid base name is still required.

#ANY - all traits with the given base name are valid targets for the bonus, unless limited using the OF or EXCEPT keywords.

#NONE – only traits with no name extension are valid targets for the bonus, unless limited using the OF or EXCEPT keywords.

Using these directives, you could target a bonus to any Animal Handling specialty except Big Cats by using gives(+1 to SK:Animal Handling (#any except "big cats")), or you could target only Big Cats by using gives(+1 to SK:Animal Handling (#none except "big cats")).

This also works with Conditionals.

There is also a special case use of #ANY to fill in for any name at all, rather than an extension, which is covered in the Bonus Targets section of Special Notes above.

SelectItemsByTag

This is a standard system used by various data file commands to select traits for processing.

> TRAITLIST, COMMANDS, TAG COMPARISON VALUE [UNLESS TAG COMPARISON VALUE]

TRAITLIST	is the list to look in, such as Skills, Equipment, or All for everything.
COMMANDS	is everything needed to specify what GCA should do with the comparison; usually, this means specifying NUM, NUMBER, NUMERIC, or # for a numeric comparison, or STRING, TEXT, or $ for a text comparison; this will vary by tag. You can also specify IGNOREEMPTY to have GCA ignore tags that are empty (this prevents ISEMPTY or ISNOTEMPTY from working; see below), and ECHOLOG if you want to fill your log with endless lines of GCA showing you what it looked at and did or did not process as a result. (You may use ECHOLOGSHORT instead of ECHOLOG. ECHOLOGSHORT will only print the command itself, and each trait that's being processed, to the log.)
TAG	is the name of the tag you want to look at. The tag name should be exactly as used by GCA.
COMPARISON	is the comparison to make. Valid comparisons are >, <, =, >=, <=, or <>, plus some special cases listed below.

Comparing text for/against an empty tag can result in unexpected behavior during text comparisons (numeric comparisons convert empty tags to 0, so use IGNOREEMPTY for those if 0 doesn't work for you in such cases), so there are two special text comparison operators that have no COMPARISON VALUE clauses: ISEMPTY and ISNOTEMPTY. ISEMPTY counts as TRUE if the specified tag is empty, and ISNOTEMPTY counts as TRUE if the tag is not empty.

There are some additional comparison-style special operators, as well: ISNUMERIC, ISNOTNUMERIC, HASINLIST, and CONTAINS. These replace the comparison operators mentioned above (>, =, etc.) and allow for checking for special text comparisons.

ISNUMERIC is true if the text can be evaluated as a number; ISNOTNUMERIC is true if the text can not be evaluated as a number; HASINLIST is true if the tag, when treated as a comma-separated list, has as one of its list values the given text; CONTAINS is true if the Tag value contains the given text.

VALUE	is the value of the tag you are looking for and should be the exact text or number to compare against.
UNLESS	is an exception block, to allow for an exception to the rule. UNLESS supports all the same comparisons as the standard clause. You create an UNLESS section by including the keyword UNLESS followed by the tag and comparison values to be made. You may also include the same Numeric or Text designations immediately after the UNLESS keyword if you want to use a different type from the base comparision.

Here are some examples:

```
#DeleteByTag Equipment, Num IgnoreEmpty, techlvl < 5
```

This command will delete any equipment items that have specified a techlvl() tag, and for which the numeric value contained in that tag is less than 5.

Note, however, that Numeric comparisons convert text values to numbers, and if there's not a number as the text value, it is probably going to convert to 0. This means that in our example here, traits such as _New Armor, that have a techlvl([techlevel]) like this, will appear to evaluate to techlvl(0), which means in our example it would be deleted. To avoid this kind of result, you'll probably need to do a TEXT comparison, and compare each techlvl you want to delete, like so:

```
#DeleteByTag Equipment, Text IgnoreEmpty, techlvl = 0
#DeleteByTag Equipment, Text IgnoreEmpty, techlvl = 1
#DeleteByTag Equipment, Text IgnoreEmpty, techlvl = 2
#DeleteByTag Equipment, Text IgnoreEmpty, techlvl = 3
#DeleteByTag Equipment, Text IgnoreEmpty, techlvl = 4
```

Be aware that in this particular example, there are a number of traits, such as _New Innate Attack, that are currently marked as techlvl(0), which you probably don't really want to delete. But, for our purposes here, this set of commands would delete everything below techlvl(5), but not delete traits with other text values.

Here is an example using UNLESS:

```
#DeleteByTag Equipment, Numeric IgnoreEmpty EchoLog, techlvl >= 5 unless $ techlvl contains ^
```

This example will delete any equipment item that has a non-empty techlvl() tag that equates to a value greater than or equal to 5, unless it contains the text ^, in which case it isn't deleted.

And here's a simple example covering all traits:

```
#DeleteByTag All, Text, x IsNotEmpty
```

This command will delete any trait where the x() tag has anything in it at all. (This would remove pretty much everything in GCA that appears with a little wrench icon next to it in the Available lists.)

Text Comparison Special Case

TAG=VALUE can use the format TAG HASINLIST VALUE or TAG CONTAINS VALUE. In the HASINLIST case, the TAG value being checked will be considered to be a comma delimited list, and the VALUE must match one of the items in that list. In the Contains case, the whole string of text in the TAG will be looked at, and if VALUE appears anywhere in the string, it will be considered to match. Both of these are case insensitive.

TAG=VALUE can also use the format TAG ISNUMERIC or TAG ISNOTNUMERIC. These test if the TAG value would qualify as a numeric value or not. While making a numeric test, remember that non-numeric values usually evaluate to 0 (or the value of any numbers beginning the text). If testing ISNUMERIC, it only qualifies as true if the entire value qualifies as a number. An example:

```
#DeleteByTag Equipment, Numeric IgnoreEmpty EchoLog, techlvl < 5 unless $ techlvl IsNotNumeric
```

A numeric test of the techlvl() tag would be done, and a number of special cases would be deleted if not saved by the UNLESS clause, which allows for preserving those values due to the fact that they're not actually numbers, just text that were converted to 0 for the simple numeric comparison.

Item Isolation or Fencing Off List Items

GCA requires that items in lists be isolated using quotes or braces if they contain a comma, since the comma is the list separator. This is often called fencing because it “fences off” the single item so that it’s not seen as multiple parts of the list. The characters used to do the fencing off are the fences: double-quotes and curly braces.

Every time GCA loads a list, then, it unfences the items in the list by removing the quotes and braces around each item. And that’s where some problems will crop up.

In operations where you might be using the product of one command as input to another, the fencing you used may have been removed, resulting in a list that is constructed without appropriate fencing as the input to the new command. GCA then starts the new operation, and the missing fencing results in incorrect processing.

This can be incredibly frustrating because it’s not immediately obvious what’s happening.

The solution is to include another set of fences. But be aware that GCA usually unfences quotes first and then braces, so in some routines you may get both sets of fences removed if you used quotes as the outer fences and braces as the inner set of fences. When in doubt, use quotes inside braces when doubling up, or use multiple layers of braces.

DOCUMENT UPDATES

August 2, 2022

The contents of the previous *GCA Data File Reference Guide* and the separate *GCA5 Updated Data File Info* documents have been combined into this document.

The content from the older *GCA Data File Reference Guide* has been updated to match the newer style used in the *GCA5 Updated Data File Info*.

Added the AddsOrIncreases() tag.

Added a Modifiers section to the Group() tag.

Added the Message() tag.

Added the Select() tag.

Added the #BuildLibraryItemList directive.

Adjusted ReachMethod in the Calculation Methods of the Gives() tag.

Added the Links() tag.

August 12, 2022

Added the DisplayCost() and DisplayWeight() tags, which were missing.

September 5, 2022

Updated the Vars() tag with naming requirements.

October 4, 2022

Removed an extraneous [, TAGLIST] from the example command templates for #BuildCharItemList, #BuildLibraryItemList, and #BuildSelectList.

November 29, 2022

Added Item Isolation or Fencing Off List Items to the Special Notes.

Added the BaseQty() tag, which has apparently been long forgotten. Swapped the order of BaseValue() and BaseWeight() tags.

Added a cross-reference from the Gives() tag topic of Bonus Classes to the Bonus Classes section of the data file.

November 30, 2022

Updated SelectX() and added the Result Variables section to it.

Added Bonus Targeting to the Name Extension Directives section of Special Notes.

Added the Rolling section to cover random roll-related functions (both numeric and text).

Updated Math Functions and Text Functions with new functions.

The Expanded Features section of Adds() has been updated with the expanded capabilities of #LOADOUT(). AddsOrIncreases() and Creates() have been updated for the same reason, but mostly just point to Adds() for more information. Loadout() has also been slightly updated for expanded features, but also mostly points to Adds() for details.

Added the SetLoadout() tag.

Added the #ASCHILD special directive to Adds(), AddsOrIncreases(), and Creates().

Expanded the explanation of the RESPOND block in Adds() to include discussion of the added ability to use a $ function to change the contents of the response.

SkillUsed() has been updated with #WORST.

Noticed that VTTNotes() and VTTModeNotes() were added at some point in the past few months without any mention here.

Added Damage Mini-Modes to the Special Notes chapter.

Added RESKIN() alias to DisplayNameFormula().

Updated Gains Bonuses in Gives().

Gives() has been updated with support for BYMODE.

Gives() and Conditional() had the LISTAS information updated.

MergeTags() and ReplaceTags() got updated for #PRESERVE.

Added #Log and #ModifierGroupWarnings to Data File Commands.

Added ShowMalf, ShowBreak, and ShowLC to the Settings section of Section Detail Information.

Added DESCRIPTION() to the Body section of Section Detail Information.

Added HitTables to the Section Detail Information.

Added the DecimalPlaces() tag.

Updated Uses() and Uses_sections() to reflect their ability to access the Solver.

Uses_settings() has been updated with info on the ALTBOX() tag and its built in styles.

December 4, 2022

Updated SelectX() with the #NOTE() and #TAGS() special directives for the LIST() tag.

Added the @SumList() function.

Added the $Modifiers() text function.

December 17, 2022

Added BaseCostFormula() and BaseWeightFormula() which were missing.

December 18, 2022

Expanded Formula() discussion on usage for equipment items.

Added CostFormula(), which is an alias for FORMULA() just for equipment items.

Added WeightFormula(), which was missing.

Started a section for Calculated Tags in Tag Detail Information and populated a bunch of equipment specific tags to it.

December 20, 2022

Added info on #FORCENEEDS to Adds() and a reference to it in AddsOrIncreases().

Added the Flexible() and CharFlexible() tags.

Added information to clarify TL() usage for equipment items.