GCA DATA FILE REFERENCE GUIDE

VERSION 3

February 24, 2012

This document is intended for the creation of files using the .GDF extension. This document covers the 3rd version of the GDF specification, as used in **GURPS®** Character Assistant 4*.*

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 4*. It may seem counter-intuitive, but this version of the GDF is version 3, not 4. 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

Throughout this document, example text that is meant to represent actual data, or examples of data, will be displayed in a different font. Examples of data as intended to be used in the data files will be shown in the different font, as well as boxed out, like so:

```
Sample Advantage, 5/10, page(sample)
```

This should make it easier to differentiate between sample items and descriptions or explanations.

Also, the text in these sample blocks will use a hanging indent, since text in this document must be wrapped, but it need not be in a data file. Please assume that when the lines are wrapped with a hanging indent, that the line is meant to be one long line, rather than the wrapped lines you see printed here.

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 considered to be 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 off of 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
AD:	for advantages
PE:	for perks
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.

FILE STRUCTURE

The data file is divided into a number of 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!

Generally speaking, 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
```

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, Advantages, Perks, 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.
*
************************************************************************
```

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.)

```
[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.

```
 [BODY]
<Humanoid>
name(Head), group(Head, All)
name(Eyes), group(Head, Eyes, All), display(-1), expanded(-1), posx(6), posy(0)
name(LeftEye), group(Head, LeftEye, Eyes, All)
name(RightEye), group(Head, RightEye, Eyes, All)
name(Neck), group(Neck, Body, Full Suit, All), display(-1), expanded(-1), posx(6), posy(50)
name(Skull), group(Head, Skull, All), basedr(2), dr(2), display(-1), expanded(-1), posx(155), posy(0)
name(Face), group(Head, Face, All), display(-1), expanded(-1), posx(155), posy(50)
name(Torso), group(Torso, Body, Full Suit, All), display(-1), expanded(-1), posx(4), posy(110)
name(Groin), group(Groin, Body, Full Suit, All), display(-1), expanded(-1), posx(195), posy(275)
name(Arms), group(Arms, Limbs, Full Suit, All), display(-1), expanded(-1), posx(200), posy(130)
name(LeftArm), group(LeftArm, Arms, Limbs, Full Suit, All)
name(RightArm), group(RightArm, Arms, Limbs, Full Suit, All)
name(Hands), group(Hands, Full Suit, All), display(-1), expanded(-1), posx(210), posy(180)
name(LeftHand), group(LeftHand, Hands, Full Suit, All)
name(RightHand), group(RightHand, Hands, Full Suit, All)
name(Legs), group(Legs, Limbs, Full Suit, All), display(-1), expanded(-1), posx(168), posy(340)
name(LeftLeg), group(LeftLeg, Legs, Limbs, Full Suit, All)
name(RightLeg), group(RightLeg, Legs, Limbs, Full Suit, All)
name(Feet), group(Feet, Full Suit, All), display(-1), expanded(-1), posx(168), posy(392)
name(LeftFoot), group(LeftFoot, Feet, Full Suit, All)
name(RightFoot), group(RightFoot, Feet, Full Suit, All)
name(Body), group(Body, Full Suit, All)
name(Full Suit), group(Full Suit, 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.

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, LeftArm, RightArm 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.

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)
```

Unlike most other trait sections, the Attributes section does not use categories for the attribute definitions.

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.

```
[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 (,).

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)
```

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 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 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 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 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 by 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.

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>**])

The tag parts and keywords are in our usual different font. The pieces that should be replaced by your data are enclosed in angle brackets and are in the **<reference font>**. 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 enabled contains within it 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 enabled 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 enabled tags, creating charX versions of the tag, then in most cases those tags also support the use of a number of special case substitutions, which function like special variables available for use within the mode enabled tag:

%curmode	replaced by the current damage mode number.
$modetag(**<tag>**)	allows access to the mode-enabled tag value, for the current mode of the **<tag>** specified.

Mode enabled tags are specified.

PRE-DEFINED TAGS

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 enabled.

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>**[, **<modetag1(data)>**][, **<modetag2(data)>**][, **<etc>**])

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

addmode({**<mode 1 name>**[, **<mode1tag1(data)>**][, **<mode1tag2(data)>**]} | {**<mode 2 name>**[, **<mode2tag1(data)>**][, **<mode2tag2(data)>**]})

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 "" 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 mod 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] [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).

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 actually 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 actually want the added traits to be added to the needs() tag of the adding trait. If #DoNotOwn is applied, #NoNeeds is not required.

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)" _
        ),
```

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 enabled.

BASECOST()

Applies to: equipment

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

BASEWEIGHT()

Applies to: equipment

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

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.

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 enabled.

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.

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; and %name% for the name of the trait granting the bonus.

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:

```
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: advantages, perks, disadvantages, quirks, templates, and 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.

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 prereq, but any item not including a listed identifier may not count the item as a prereq.

An item may now 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().

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] [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 actually 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 actually want the added traits to be added to the needs() tag of the adding trait. If #DoNotOwn is applied, #NoNeeds is not required.

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

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 enabled.

DAMAGEBASEDON()

Applies to: traits with damage modes

If this tag exists, GCA will use the stat 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 stat specified within the function.

This tag is mode enabled.

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 the end if available.

This tag is mode enabled.

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 enabled.

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.

DESCRIPTION()

Applies to: traits and modifiers

This tag includes a description of the trait.

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 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 has to 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.

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.

DOWNTO()

Applies to: traits other than equipment

This tag is math enabled.

Advantages, Perks, Disadvantages, Quirks, and Templates

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.

DB()

Applies to: traits

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

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.

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: advantages, perks, disadvantages, quirks, templates, and 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.

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.

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 **<target>**[::**<tag>**][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.
<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.
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; and %name% for the name of the trait granting the bonus.

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)
```

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 ::**<tag>** 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: ::**<tag>**$. For example, to apply a text bonus to the damtype() tag of a trait, you might use an expression like gives(=" dbk" to owner::damtype$), 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 characters 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**
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
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
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().
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	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

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.

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.

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.

IDENT()

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: advantages, perks, disadvantages, and quirks

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.

LC()

Applies to: traits with damage modes

The LC value of the weapon or attack.

This tag is mode enabled.

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: advantages, perks, disadvantages, quirks, and modifiers

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 a name for the level 0 value of the trait.

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>**"[, 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.
<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.

MINSCORE()

Applies to: attributes

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

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 enabled.

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 to do the check.

This tag is mode enabled.

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:

(1) mitigator(yes) will set mitigated(yes), which means bonuses will be changed to conditional bonuses, and taboos will be ignored;
(2) mitigator(taboo) will set mitigated(taboo), which means taboos will be ignored, but bonuses will remain bonuses;
(3) 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 enabled.

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 a large number of 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 <category name> [= **<value>**]

Some number of traits is required, from the category **<category name>**, 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.

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()

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 enabled.

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 enabled.

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 enabled.

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 enabled.

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 enabled.

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 enabled.

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 enabled.

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.

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>**”[, 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.
<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.

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 enabled.

ROUND() (ATTRIBUTES)

Applies to: attributes

This tag specifies a value that determines how the attribute score should be rounded off. If the value is less than zero, the attribute will be rounded down; if the value is greater than zero, the attribute will be rounded up.

This tag is math enabled.

ROUND() (MODIFIERS)

Applies to: modifiers

This tag specifies how this modifier’s effects should be rounded off. The default is to round values up, but if round(down) is specified, then rounding will be down.

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.

SELECTX()

Applies to: traits

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.

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 Mods, and adjust their levels and such.

```
Select<X>(Text(<dialog text>),
            MultiType(yes),
            PointsWanted([ exactly | upto | atleast ] <cost>),
            ItemsWanted([ exactly | upto | atleast ] <number> [, [ exactly | upto | atleast ] <number>] ),
            TagWith(<tag values>),
            List(<trait>[, #newitem(<definition>)][#codes(<codes>)])  )
```

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

Select**<X>** 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.

ItemsWanted([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 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.

PointsWanted([exactly | upto | atleast] **<cost>**) 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.

TagWith(**<tag values>**) allows you to tag any traits added/modified by the SelectX() to be tagged with one or more tags and values after OK is clicked on the SelectX dialog.

List(**<trait>**[, #newitem(**<definition>**)][#codes(**<codes>**)]) 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.
- #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 (applicable only to skills and spells). If using both, separate with a comma.

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)_
                              )_
          )_
)_
)
```

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.

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 enabled.

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.

This tag is mode enabled.

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.

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.

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.

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).

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.

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.

UPTO()

Applies to: traits

This tag is math enabled.

Advantages, Perks, Disadvantages, Quirks, and Templates

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.

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 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.

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.

@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>**.

@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.

@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(**<itemname>**, **<modname>**)

Where:

<itemname>	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.

<itemname> and **<modname>** may be enclosed in quotes or braces, and should be if either might contain a comma.

@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(**<value list>**)

Where:

<value list>	is the list of values that are to be evaluated, separated by commas. Each value may be an expression.

The **<value list>** can be as long as required.

@MIN

This function returns the minimum value from a list of values.

@min(**<value list>**)

Where:

<value list> is the list of values that are to be evaluated, separated by commas. Each value may be an expression.

The **<value list>** 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.

@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.

@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 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.

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.

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.

$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 returning either **<result>** or **<altresult>**, depending on the result of **<expression>**.

$if(**<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.

<result> and **<altresult>** may be enclosed in quotes or braces, and should be if either might contain a keyword.

$If does not currently support the use of elseif.

$INDEXEDVALUE

This function allows for returning a text 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 text values that follows.
<value list>	is the list of text values that are to be returned, separated by commas, and enclosed in quotes or braces as required.

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.

$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 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.

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>** **<comparison>** **<tagvalue>** , template([**<some text>**]%ListItem%[**<more text>**]) [, **<taglist>**])

#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 word or 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.
<comparison>	[is | isnot | includes | excludes | listincludes | listexcludes] 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 with Hobbies 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%[**<some 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.

#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%[**<some 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%[**<some 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.

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 SelectX() specific directive to allow for building a list of items for use in the SelectX() item list, which is processed when the SelectX() becomes active. This means it may possibly include items from earlier SelectX() picks, which wouldn't be the case for normal directives.

#BuildSelectList(**<Type>** where **<tag>** **<comparison>** **<tagvalue>** , template([**<some text>**]%ListItem%[**<more text>**]) [, **<taglist>**])

#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 word or 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.

<comparison> [is | isnot | includes | excludes | listincludes | listexcludes] 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 with Hobbies 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%[**<some 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:

```
select3(_
    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 SelectX(), 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 SelectX(), 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 SelectX() statement above would look like this immediately before it was processed by the selection window, as if you'd written it this way:

```
select3(_
    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)_
    )_
)
```

#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 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.

Examples:

```
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(<item list>),
             Name(<choice name>),
             Title(<dialog title>),
             Text(<dialog text>),
             AliasList(<alias list>),
             altXlist(<alt X list>),
             Default(<initial selections list>),
             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(**<item list>**) 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(**<choice name>**) 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(**<dialog title>**) allows you to specify the title, or caption, of the dialog displayed to the user.

Text(**<dialog text>**) allows you to specify the text explaining the purpose of the dialog displayed to the user.

AliasList(**<alias list>**) 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(**<alt X list>**) allows you to specify up to 5 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 5, in place of the X in the tag name. For example, alt1list() or alt2list().

Default(**<initial selections list>**) 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().

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.

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.

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.

%**<choice name>**% GCA will insert the user's first selected option in place of this variable.

%**<choice name>**cost% GCA will insert the specified cost of the user's first selected option in place of this variable.

%**<choice name>**alias% GCA will insert the AliasList() item corresponding to the user's first selected option in place of this variable.

%**<choice name>**#% 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.

%**<choice name>**cost#% 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.

%**<choice name>**#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.

%**<choice name>**#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.

%**<choice name>**list% GCA will insert a comma separated list of all the user’s selected options in place of this variable.

%**<choice name>**costlist%GCA will insert a comma separated list of all the costs corresponding to the user’s selected options, in place of this variable.

%**<choice name>**aliaslist% GCA will insert a comma separated list of all the AliasList() items corresponding to the user’s selected options, in place of this variable.

%**<choice name>**altXlist%GCA will insert a comma separated list of all the altXlist() items corresponding to the user’s selected options, in place of this variable.

%**<choice name>**notlist% GCA will insert a comma separated list of all the list() items *not* chosen by the user, in place of this variable.

%**<choice name>**costnotlist% GCA will insert a comma separated list of all the costs corresponding to items *not* chosen by the user, in place of this variable.

%**<choice name>**aliasnotlist% 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 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.

#EDIT

The #Edit directive prompts GCA to offer the user the Edit window, with which they can immediately make adjustments to the trait.

#Edit

This directive has no options, and therefore has no parentheses or available tags.

#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 something, if 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>**,] **<target text>**)

or

#InputReplace(**<prompt>**, **<target text>** [, **<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.
<target text>	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 **<target text>** should appear in the tag list somewhere.
<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>**, **<target tag>** [, **<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.
<target tag>	is the tag whose value should become the text entered by the user. This value should be enclosed in quotes or braces if necessary. Note that to be effective, the **<target text>** should appear in the tag list somewhere.
<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 **<target tag>** already has a value, #InputToTag will be skipped, and no value 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>**, **<target tag>** [, **<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.
<target tag>	is the tag whose value should become the text entered by the user. This value should be enclosed in quotes or braces if necessary. Note that to be effective, the **<target text>** should appear in the tag list somewhere.
<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 **<target tag>** 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 list. 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 list. 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: #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.

Special Exemption: Unlike other directives, #message remains in the trait, so if it is duped-n-pasted to another trait, the message will trigger again.

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.

#CLONE

From GCA wiki, by Foxfire; modified.

This command allows for duplicating an existing trait as a new trait.

#Clone "**<trait>**" as "**<newtrait>**"

<trait>	is the name of the existing trait to be duplicated. The full name, including prefix tag, should be specified.
<newtrait>	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.

#DELETE

From GCA wiki, by Foxfire; modified.

This command allows for deleting a trait from the loaded data.

#Delete "**<trait>**"

<trait>	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.

#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 of EchoLog. 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 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.

#DELETEFROMGROUP

This command allows for removing Group items from a group.

#DeleteFromGroup **<GroupName> <SK:Whatever>** [, **<SK:Whatever (Ext)>**][, ALL **<SK:Buncha Skills>**]

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.

#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 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 included, 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).

#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.
<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.

CHANGES

February 10, 2012

Initial release.

February 24, 2012

Expanded description of Data File Commands.

Expanded description of Directives.

Slightly updated Custom Tags.

Special Case Equipment Targets added to Target Tag Bonuses in gives().

Added tags: age(), appearance(), bodytype(), charheight(), charweight(), race()

Added tags: mergetags(), replacetags()