GCA5 UPDATED DATA FILE INFORMATION

FOR GCA5 RELATED UPDATES TO THE GDF FILE SPEC

Last Updated: April 20, 2020

This document is intended for the creation of files using the .GDF extension. This document covers extensions to the GDF specification, as used in **GURPS**® Character Assistant 5*.*

CHANGES

2017 December 19

Through b101.

General Information

- Added Bonus Targets
- Added the FE: prefix tag for Features to Prefix Tags

Section Detail Information

- Added Attributes
- Added Features
- Added Bonus Classes

Tag Detail Information

- Added Bulk()
- Added DRNotes()
- Added Default()
- Added Features()
- Added Enhanced Parsing Changes to Gives()
- Added Bonus Classes to Gives()
- Added ItemNotes()
- Added Round()
- Added STCap()

Data File Commands

- Added information to #ChoiceList

Special Notes

- Added Modes

2020 April 20

Tag Gives()

- Updated the table for Calculations in the Target Tag Bonuses subsection

CONVENTIONS

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

> [Example]
> This is an example.

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

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

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

These templates often include extraneous spaces to clarify symbols used.

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

General Information

Prefix Tags — Expanded

CU:	for Cultural Familiarities
FE:	for Features
LA:	for Languages

Bonus Targets — New

This content was apparently overlooked in previous documents, so I've added it here.

When granting bonuses, knowing the right target is important, and the right target isn't always a trait. To grant bonuses to items that aren't specifically individual traits, here are the targets you can use.

By Prefix

When you want to grant a bonus to a target based on its category (specific by trait type), use one of the prefixes given here.

CA:	to a category of Advantages
ADCAT:	to a category of Advantages
DICAT:	to a category of Disadvantages
DISADCAT:	to a category of Disadvantages
LANGCAT:	to a category of Languages
LACAT:	to a category of Languages
CULTCAT:	to a category of Cultural Familiarities
CUCAT:	to a category of Cultural Familiarities
FEATURECAT:	to a category of Features
FECAT:	to a category of Features
PECAT:	to a category of Perks
PERKCAT:	to a category of Perks
EQCAT:	to a category of Equipment
EQUIPCAT:	to a category of Equipment
SKCAT:	to a category of Skills
SKILLCAT:	to a category of Skills
CL:	to a category of Skills (a Class of Skills)
SPCAT:	to a category of Spells
SPELLCAT:	to a category of Spells
CO:	to a category of Spells (a College of Spells)
GR:	to a Group (defined data file Group, not a modifier group)

By Keyword

When you want to target a bonus to specific things based on some pre-defined targets.

SKILLS	to all Skills (levels)
SKILLPOINTS	to all Skills (points)
SPELLS	to all Spells (levels)
SPELLPOINTS	to all Spells (points)
CULTURES	to all Cultural Familiarities
CULTURE	to all Cultural Familiarities

LANGUAGES	to all Languages
LANGUAGE	to all Languages
FEATURES	to all Features
FEATURE	to all Features

STSKILLS	to all Skills based on ST (type ST/?; levels)
DXSKILLS	to all Skills based on DX (type DX/?; levels)
IQSKILLS	to all Skills based on IQ (type IQ/?; levels)
HTSKILLS	to all Skills based on HT (type HT/?; levels)
STSKILLPOINTS	to all Skills based on ST (type ST/?; points)
DXSKILLPOINTS	to all Skills based on DX (type DX/?; points)
IQSKILLPOINTS	to all Skills based on IQ (type IQ/?; points)
HTSKILLPOINTS	to all Skills based on HT (type HT/?; points)

STSPELLS	to all Spells based on ST (type ST/?; levels)
DXSPELLS	to all Spells based on DX (type DX/?; levels)
IQSPELLS	to all Spells based on IQ (type IQ/?; levels)
HTSPELLS	to all Spells based on HT (type HT/?; levels)
STSPELLPOINTS	to all Spells based on ST (type ST/?; points)
DXSPELLPOINTS	to all Spells based on DX (type DX/?; points)
IQSPELLPOINTS	to all Spells based on IQ (type IQ/?; points)
HTSPELLPOINTS	to all Spells based on HT (type HT/?; points)

Section Detail Information

Body Images NEW

GCA now supports assigning custom images for body types.

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

A BodyImages block might look like this:

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

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

GCA will search for the images in the following order:

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

Flag Symbols NEW

AKA those little icons marking supernatural, exotic, etc.

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

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

There are three sections per line, separated by commas:

> NAMEOFSYMBOL , IMAGEFILE , CRITERIA

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

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

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

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

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

Commands

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

#Clear

```
#CLEAR
```

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

#Delete

```
#DELETE LISTOFNAMES
```

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

```
#Delete Mental Advantage, Mental Disadvantage
```

Criteria

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

The basic criteria specification looks like this:

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

such as

> Ads where cat listincludes Mental

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

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

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

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

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

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

> TAG compare TAGVALUE

OR

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

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

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

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

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

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

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

Attributes EXPANDED

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

Cultural Familiarities NEW

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

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

Languages NEW

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

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

Features NEW

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

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

Bonus Classes NEW

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

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

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

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

Here's how you define entries for the BonusClasses block in the data files:

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

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

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

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

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

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

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

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

For example:

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

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

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

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

See also Bonus Classes in Gives().

TAG DETAIL INFORMATION

Pre-Defined Tags

Adds() EXPANDED

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

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

Base() NEW

Applies to: advantages, perks, disadvantages, quirks

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

```
base(10)
```

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

Math enabled.

Bulk() EXPANDED

Now mode-enabled, and will create a charbulk() tag when the trait is added to the character.

Creates() EXPANDED

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

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

Default() EXPANDED

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

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

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

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

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

@Default()

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

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

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

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

Deflect() NEW

Applies to: traits

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

DisplayName() NEW

Applies to: system traits

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

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

DisplayNameFormula() NEW

Applies to: traits, modifiers

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

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

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

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

DisplayScoreFormula() NEW

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

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

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

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

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

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

DownFormula() NEW

Applies to: attributes

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

DRNotes() NEW

Applies to: traits

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

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

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

DownTo() EXPANDED

Applies to: traits other than equipment, modifiers

Now applies to modifiers.

Features() NEW

Applies to: templates

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

FencingWeapon() NEW

Applies to: traits

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

> fencingweapon(char::enclevel)

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

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

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

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

Fortify() NEW

Applies to: traits

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

Gives() EXPANDED

Applies to: traits, modifiers

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

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

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

There are two ways to create a bonus exception:

1) **Unless**

Create the exception using the new UNLESS keyword:

```
UNLESS TARGET[::TAG] = Z
```

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

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

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

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

2) OnlyIf

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

```
ONLYIF TARGET[::TAG] = Y
```

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

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

Targets

Bonuses can now be granted TO CULTURES or TO LANGUAGES.

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

```
TO CUCAT: CATEGORY
```

or

```
TO LACAT:CATEGORY
```

See also the Bonus Targets *block in the* General Information *section.*

User Targetable Bonuses (%ChosenTarget%)

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

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

```
gives(+1 to %chosentarget%)
```

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

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

Gains Bonuses (the From Keyword)

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

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

Note that this type of bonus still comes from the GIVES() tag, but requires the use of the FROM keyword, instead of the usual TO keyword (explicit or implied). This structure implies a TO ME clause, but one is not supported.

For example, the tag

> gives(+1 from AD:Magery)

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

The 'gains' feature should support the expanded keywords such as ONLYIF and UPTO and whatnot, but does **not** currently support drawing from specific tags of the given source.

Target Tag Bonuses

Traits may now have the LOCATION() tag targeted.

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

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

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

Calculation Methods (Updated)

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

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

Target Tag	Stored To	Modes	Method Used
acc	characc	X	VALUEMETHODSUFFIX
armordivisor	chararmordivisor	X	VALUEMETHOD
blockat	blocklevel		SCOREMETHOD
break	charbreak	X	VALUEMETHOD
bulk	charbulk	X	VALUEMETHOD
damage	chardamage	X	Special.
damtype	chardamtype	X	Special. Do NoBase, Append Bonus String, Damage Mode Special Case Subs, Text Function Solver
db	chardb		Simple math; no text bonuses supported
deflect	chardeflect		VALUEMETHOD
dr	chardr		Simple math; no text bonuses supported, suffix preserved
effectivest	chareffectivest	X	Special. ST=DamageBasedOn, Do NoBase, Do NoCalc, Append Bonus String, Solver, Apply Bonuses
fencingpenalty	charfencingpenalty		VALUEMETHOD
fortify	charfortify		VALUEMETHOD
level	level		Special.
location	charlocation		Special. Text and text functions only.
minst	charminst	X	VALUEMETHODSUFFIX
parry	charparry	X	Special. Do NoBase, Append Bonus String, Damage Mode Special Case Subs, TextFunctionSolver, “no”, U/F suffix perserved, Solver, Apply Bonuses
parryat	parrylevel, parryatbonus, parryatmult		SCOREMETHOD
parryscore	charparryscore	X	Special. There is no base parryscore() tag; bonuses targeted to it are used in the calculation of charparryscore() based on charskillused(), charparry(), parryat(), parryatbonus(), and parryatmult().
points	points		Special.
radius	charradius	X	RANGEMETHOD
raiseruleof	raiseruleof		Special. There is no base raiseruleof() tag; bonuses targeted to it are stored in raiseruleof(); only +/- bonuses are supported
rangehalfdam	charrangehalfdam	X	RANGEMETHOD
rangemax	charrangemax	X	RANGEMETHOD
reach	charreach	X	REACHMETHOD
shots	charshots	X	VALUEMETHODSUFFIX
skillscore	charskillscore	X	Special. There is no base skillscore() tag; bonuses targeted to it are used in the calculation of charskillscore() when evaluating skillused()

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

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

REACHMETHOD 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.

Bonus Classes

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

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

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

Enhanced Parsing Changes

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

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

or you could use

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

and either should work correctly and without issue.

Spaces around the keywords aren't required any more, except for around the TO and FROM keywords in the basic bonus block, but you must enclose other blocks in quotes or braces if

they might include any of the other keywords: ONLYIF, UNLESS, LISTAS, CLASS, UPTO, and WHEN. GCA will strip the containers after parsing. (Parsing should now honor braces as well as quotes as containers for these clauses.)

Invisible() NEW

Applies to: system traits

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

```
invisible(yes)
```

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

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

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

ItemNotes() NEW

Applies to: traits

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

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

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

Mode-enabled.

Loadout() NEW

Applies to: equipment

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

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

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

For example

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

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

MergeTags() EXPANDED

Expanded to allow targeting 'char' or 'character' with tag values. For example:

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

ReplaceTags() EXPANDED

Expanded to allow targeting 'char' or 'character' with tag values. For example:

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

Removes() NEW

Applies to: templates

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

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

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

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

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

RemovesByTag() NEW

Applies to: templates

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

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

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

Note that it is very dangerous to use criteria that specifies a non-match, because that means any trait without the specified tag will likely not match, and will be removed. REMOVESBYTAG() is

intended to be used primarily in conjunction with tag values added during a SELECTX() process, for further customizing templates with lenses later on, but data file authors will do as they will.

Round() EXPANDED

Applies to: traits, modifiers

Standardized a bit for more consistency across attributes and modifiers.

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

Select(), SelectX() EXPANDED

Applies to: templates

GCA will now reject #NEWITEM() commands that don't include the proper prefix tags.

Conditional()

Added support for the new CONDITIONAL() tag in the SELECT() structure. The CONDITIONAL(), if it exists, must evaluate to True (a non-zero result) in order for the SELECT() to be processed.

```
triggers(_
        select(_
                text("Select with Conditional. You'll never see this one."),
                conditional(@sametext("alpha", "bravo")),
                itemswanted(atleast 1),
                list(_
                        DI:Alcoholism,
                        DI:Stubbornness,
                        DI:Workaholic_
                        )_
                )_
        )
```

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

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

AddMods(), RemoveMods()

Items being referenced by the EXISTING() tag may now include ADDMODS() and/or REMOVEMODS() in the specified tag list, in order to have those tags applied to the EXISTING() trait.

MultiType()

Now supports the use of MIXEDTYPE(YES) as a synonym for MULTITYPE(YES).

STCap() NEW

Applies to: traits

The way GCA supports the maximum ST cap for muscle-powered weapons has changed. GCA will now apply a ST cap to all muscle-powered weapons, regardless of type.

Because it's now easier for the cap to be applied to weapons that perhaps aren't actually meant to be capped (either by RAW or in your campaign), GCA now supports the STCAP() tag to turn off capping, or to change the multiplier value if desired.

Use STCAP(NO) to turn off the max ST cap:

```
stcap(no)
```

or use STCAP(X), replacing X with a value, to replace the 3*ST cap with an X*ST cap:

```
stcap(2)
```

Math-enabled. Mode-enabled.

SubsFor() NEW

Applies to: traits

This tag supports the new Substitutions system, which allows one trait to substitute for another when GCA is tracking down references. This system applies to character traits, and does not do substitutions on a library basis.

```
SUBSFOR( TRAITNAME [, TRAITNAME [, ... ] ])
```

TRAITNAME must be a fully qualified trait name, including a prefix specifying the trait type. Multiple traits may be defined by separating them with commas. Traits should be enclosed in quotes or braces as required.

See the Substitutions section in Special Notes for more information.

TargetListIncludes() NEW

Applies to: traits

This tag supports the new "user targetable" bonuses feature, to specify the types of target traits that are applicable to a trait that supports such bonuses.

File authors can suggest and limit the available targets presented to the user by using the TARGETLISTINCLUDES() tag on the item, which includes one or more criteria for adding traits to the list, using a structure similar to that in the #BUILDSELECTLIST directive. It's structured like this:

```
TARGETLISTINCLUDES( TRAITTYPE [ WHERE TAG [ IS | ISNOT | INCLUDES | EXCLUDES | LISTINCLUDES |
LISTEXCLUDES ] TAGVALUE ][ , ... ] )
```

which allows for specifying just the trait's type in TRAITTYPE if that's the only limit required. If more restriction is needed, see the docs for #BUILDSELECTLIST to see how the WHERE block works. Note: GCA will never include locked or hidden traits as valid targets. In addition, TRAITTYPE may be ANY or ALL to specify selecting from all traits, not just one type.

As an example, instead of requiring users to add a Weapon Master Damage Bonus modifier to every weapon that gets the bonus, the bonus can be included with the Weapon Master advantage, and can use the new %CHOSENTARGET% target instead. The user can then select or modify the weapons that apply whenever they need to do so in the Edit window.

Here's a modified version of Weapon Master:

```
Weapon Master (Targets), 20/25/30/35/40/45, levelnames(one specific weapon, two weapons normally used
together, a small class of weapons, a medium class of weapons, a large class of weapons, all muscle
powered weapons), page(B99), upto(6), cat(Mundane, Physical),
      x(_
            #InputToTagReplace("Please specify the weapon, weapons, or class of weapons you have
Mastery of:", nameext, , "Weapon Master")_
            ),
      gives(_
            =+@if(_
                  $modetag(charskillscore) = ST:DX+1 _
                  THEN @textindexedvalue($modetag(dmg), ("thr", char::basethdice), ("sw",
char::baseswdice), ELSE $solver(me::dmg)) _
                  ELSE @if(_
                        $modetag(charskillscore) > ST:DX+1 _
                        THEN @textindexedvalue($modetag(dmg), ("thr", 2 * char::basethdice),
("sw", 2 * char::baseswdice), ELSE 2*$solver(me::dmg)) _
                        ELSE 0 _
                        )_
                  ) to %chosentarget%::damage$ listas Weapon Master Damage Bonus _
            ),
      targetlistincludes(Equipment where charreach isnot "", Equipment where charrangemax isnot "")
```

Triggers() NEW

Applies to: templates

GCA now supports a more flexible means of handling a variety of template features.

The TRIGGERS() tag is a wrapper/processor tag that contains all of the various "command" tags, or tags that are "triggered" by an item being added to a character. This new tag allows for all of those trigger tags to be included and processed in the order desired.

In addition, you may include any number of each tag that you may wish. That means you could include an ADDS() then a REMOVES() and then another ADDS(). You may also include SELECT() tags, which no longer need to be numbered sequentially; they'll be processed in the order they're included within the TRIGGERS() tag, as they're encountered. (If you do number them, GCA will ignore the numbers and still process them in the order listed.)

The syntax for each of the trigger tags within TRIGGERS() remains exactly the same.

The existing system for the trigger tags still remains, and still works in the exact same fashion it always has, with the exact same processing order, and the same limitation of one tag of each type. (And for that system, you must still use the sequentially numbered SELECTX() tags, as well.)

The new TRIGGERS() tag contents will be processed first, **before** any of the old system tags are processed.

Here are all the trigger tags currently recognized by GCA:

ADDMODS(), ADDS(), BODYTYPE(), CHARHEIGHT(), CHARWEIGHT(), CHILDOF(), CREATES(), LOADOUT(), MERGETAGS(), PARENTOF(), RACE(), REMOVEMODS(), REMOVES(), REMOVESBYTAG(), REPLACETAGS(), SELECTX(), SELECT(), SETS().

Processing of the various tags within TRIGGERS() supports Text Function Solver processing of each tag's contents before it is evaluated. This means, for example, that you could use an $IF() function to dynamically allow for different possible versions of the same tag, based on other values.

In addition, processing of the various sub-tags also supports #BUILDSELECTLIST processing of each tag's contents. In most cases, #BUILDSELECTLIST isn't necessary, or even makes no sense at all. However, it does allow for one way of dynamically changing the intended content of trigger tags based on previous activity in the TRIGGERS() tag sequence.

UpFormula() NEW

Applies to: attributes

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

Vars() NEW

Applies to: traits

GCA now supports defining variables at the trait level. Variables are defined using the VARS() tag, like so:

```
VARS( NAME1 = VALUE1 [, NAME2 = VALUE2 ] [, ... ] )
```

This allows for greatly reducing the complexity of certain types of formulas, such as the new DISPLAYNAMEFORMULA().

These are simple substitution variables; if you were to use

```
vars(%name% = me::name)
```

then the %name% variable stores the text me::name, **not** the actual name of the trait.

For example, to create a DisplayName that uses the same name that GCA would generate, but adds the additional text fragment Bonus Text **within** the parenthetical information (if any), the straight formula might look like this:

```
displaynameformula($if(@endswith($val(me::basedisplayname), %closeparen) then
$insertinto($val(me::basedisplayname), "; Bonus Text", @len($val(me::basedisplayname))-1) else "(Bonus
Text)" ))
```

Notice that we have to keep repeating the $val(me::basedisplayname) bit over and over again. (And $VAL() is necessary, because the Text Function Solver won't replace any value references unless

explicitly told to do so with the $VAL() or $TEXTVALUE() functions). Just imagine the additional complexity if we wanted to include a trait value or tag reference instead of the simple constant Bonus Text.

So, we can simplify with a variable:

```
vars(%name% = $val(me::basedisplayname))
```

and the new DISPLAYNAMEFORMULA() that makes use of it:

```
displaynameformula($if(@endswith(%name%, %closeparen) then $insertinto(%name%, "; Bonus Text",
@len(%name%)-1) else "%name% (Bonus Text)" ))
```

In this case, not drastically shorter as text goes, but much more readable.

Note that because GCA will replace the variable name indiscriminately within the target area, as the first step of evaluating an expression, you should ensure that your variable names are unlikely to conflict with other types of text. I recommend using a percent sign % at the beginning and end of each variable name, to ensure no accidents are likely to occur.

Where() NEW

Applies to: equipment

The WHERE() tag is in support of 'Where It's Kept' on the character, since LOCATION() is used for armor coverage. This is for those who like to detail out where each item is placed on their character. (Now supported for editing on character equipment in the Simple Edit window.)

MATH

Math Functions

@Ceiling, @Ceil — NEW

```
@CEILING(VALUE)
@CEIL(VALUE)
```

Returns the smallest integer value that is greater than or equal to the given VALUE.

@EndsWith — NEW

```
@ENDSWITH( DOESTHIS, ENDWITHTHIS )
```

Returns 1 if DOESTHIS ends with ENDWITHTHIS, 0 if not. Case is ignored.

If you want to check for parens ( ), braces { }, or quotes ", you can instead use these variables as needed: %CLOSEPAREN, %OPENPAREN, %CLOSEBRACE, %OPENBRACE, %QUOTES.

Note that the %CLOSEPAREN, %OPENPAREN, %CLOSEBRACE, and %OPENBRACE special variables should not be necessary if the corresponding characters are enclosed within quotes in the function, such as

```
@endswith(")")
```

The %QUOTES variable is still required, as there is no way to enclose the double quote character safely, but it's also the least likely to be needed for anything.

@Floor — NEW

```
@FLOOR(VALUE)
```

Returns the largest integer value that is less than or equal to the given VALUE.

@HasMod — EXPANDED

Will now look for modifiers that have name extensions using either the old NAME (NAME EXT) format, or the newer NAME, NAME EXT format that modifiers display in the UI now.

@ItemHasMod — EXPANDED

Will now look for modifiers that have name extensions using either the old NAME (NAME EXT) format, or the newer NAME, NAME EXT format that modifiers display in the UI now.

@Len — NEW

```
@LEN(TEXT)
```

Returns the length of the given TEXT.

@OwnerHasMod — EXPANDED

Will now look for modifiers that have name extensions using either the old NAME (NAME EXT) format, or the newer NAME, NAME EXT format that modifiers display in the UI now.

@StartsWith — NEW

> @STARTSWITH(DOESTHIS, STARTWITHTHIS)

Returns 1 if DOESTHIS starts with STARTWITHTHIS, 0 if not. Case is ignored.

If you want to check for parens ( ), braces { }, or quotes ", you can instead use these variables as needed: %CLOSEPAREN, %OPENPAREN, %CLOSEBRACE, %OPENBRACE, %QUOTES.

Note that the %CLOSEPAREN, %OPENPAREN, %CLOSEBRACE, and %OPENBRACE special variables should not be necessary, if the corresponding characters are enclosed within quotes in the function, such as

> @startswith(")")

The %QUOTES variable is still required, as there is no way to enclose the double quote character safely, but it's also the least likely to be needed for anything.

@TextIsInList, @TextIsInListAlt — NEW

Two similar functions to determine if a given text snippet can be found within a list of text items.

> @TEXTISINLIST(ISTHIS, { INTHISITEMLIST })
> @TEXTISINLISTALT(ISTHIS, USINGTHISSEPARATORCHARACTER, { INTHISITEMLIST })

Both functions are similar, except that @TEXTISINLIST assumes a comma separated list of items, as is used in the CAT() or PAGE() tags, while @TEXTISINLISTALT allows you to specify the character used for separating the items in the list, such as the pipe (|) character used in SKILLUSED().

You should always enclose the list parameter inside braces to ensure you are safely defining the list of items as a single unit. You can, of course, either specify the list items manually, or obtain them using something like $TextValue(me::cat).

Comparison of items ignores case, as always. The value returned is 0 if the item was not found, or the index number (1-based) of the list item that was matched.

@TextIsInText — NEW

> @TEXTISINTEXT(ISTHIS, WITHINTHIS)

Returns 1 if the ISTHIS text is contained within the WITHINTHIS text, and 0 if not. As with most such things in GCA, case is ignored.

@TotalChildrenTag — NEW

Totals up the values of the specified tag from all children, and returns it.

> @TOTALCHILDRENTAG(TAG [#NOMULT])

Note that children only are viewed; it does not recurse through nested children. It does, however, multiply the tag value by the child's COUNT to get the total value. If you don't want the value multiplied by COUNT, include the #NOMULT directive in the function call.

@TotalOwnerChildrenTag	NEW

Same as @TOTALCHILDRENTAG but meant to be used from a modifier, to get the owning trait's children.

@TOTALOWNERCHILDRENTAG(TAG [#NOMULT])

TEXT PROCESSING

Text Functions

$If — EXPANDED

Now supports ElseIf blocks.

```
$IF( EXPRESSION1 THEN RESULT [ ELSEIF EXPRESSION2 THEN RESULT2 [ ... ] ] [ ELSE LASTRESULT ] )
```

$InsertInto — NEW

Inserts some text into some other text.

```
$INSERTINTO( INTOTHIS, PUTTHIS, ATPOSITION )
```

INTOTHIS is the text that is going to receive the PUTTHIS, and the text is inserted at position ATPOSITION.

$Val — NEW

This is identical in function to $TEXTVALUE(), just shorter.

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.

#BuildList — EXPANDED

Now supports more than one instance of the %LISTITEM% variable in the output template.

#DeleteMe — NEW

Instructs GCA to delete the item once it's been fully processed.

This may seem a bit odd, but it allows for creating templates that add things to the character and are then removed, so that the user doesn't have to manually remove them later when they no longer serve any purpose. (Note some users may find it confusing if a trait doesn't appear in the Character list after they've added it, so this directive is mostly for those that want to use it in their custom data.)

#Format — NEW

Modifiers now support a new directive, #format, within the shortname() tag, which tells it to format the output in a particular way—pretty much just like displaynameformula(), but obviously set up a tad differently.

It is used like so

```
shortname( #format $val(me::levelname) )
```

#Format must be the first part of the tag value, and everything that follows is the format/formula to use. If you need spaces on one end or another, enclose the text after the #format directive in quotes or braces.

Combined with DisplayNameFormula(), this means that you can now usefully create all-in-one modifiers, and don't need to create a separate modifier for each different level of effect. For example:

```
[MODIFIERS]
<Self-Control>
Self-Control Roll, *0.5/*1/*1.5/*2, upto(4), downto(1), group(Self-Control), page(B121),
      shortname(#format $val(me::shortlevelname) ),
      levelnames("15 or less, almost all the time",
                 "12 or less, quite often",
                 "9 or less, fairly often",
                 "6 or less, quite rarely"),
      shortlevelnames(15 or less,
```

```
                    12 or less,
                    9 or less,
                    6 or less),
    displaynameformula( You resist on a roll of $val(me::levelname) )
```

(I set up this example in order of increasing costs, which is in decreasing order of Roll value. It could, of course, also be set up for the reverse order, instead.)

Inside GCA:

- the user sees **Self-Control Roll** in the Available Modifiers list
- when added to the character they see **You resist on a roll of 15 or less, almost all the time** in the Applied Modifiers list
- and for the caption within the item using this modifier, they see **15 or less, *0.5**.

If the user then increments the modifier, the values change appropriately to the next levels.

Further, if you need some item to check for the existence of this modifier with @HASMOD(), you can simply use @HasMod(Self-Control Roll), rather than needing to check for the name of each individual level, as is the case now with each level being a separate modifier.

Other examples:

```
[MODIFIERS]
<Chronic Pain>
Interval, *0.5/*1/*1.5/*2, upto(4), downto(1), shortname( #format $val(me::levelname) ),
      levelnames(1 hour, 2 hours, 4 hours, 8 hours), group(Chronic Pain), page(B126),
      displaynameformula( $val(me::name): $val(me::levelname) )

Frequency, *0.5/*1/*2/*3, upto(4), downto(1), shortname( #format $val(me::levelname) ),
      levelnames(6 or less, 9 or less, 12 or less, 15 or less), group(Chronic Pain), page(B126),
      displaynameformula( $val(me::name): Attack occurs on a roll of $val(me::levelname) )
```

#Ref NEW

Provides a way to reference modifiers, rather than defining them, in various functional tags.

Supported in INITMODS(), ADDMODS(), ADDS(), CREATES().

InitMods() EXPANDED

#REF allows for specifying modifiers by reference in INITMODS(), and other places, so that an entire modifier definition no longer needs to be included. The existing behavior is preserved, but now you can also specify a reference to a modifier instead, using the #REF directive and this format:

```
INITMODS( #REF MODIFIERNAME [ = X ]  [ FROM GROUPNAME ] )
```

Braces are optional if not needed to protect parsing on the = or FROM keywords, or the pipes separating various INITMODS() items. (If the modifier has a name extension, you do need to include it inside parens as part of MODIFIERNAME, as per usual.)

The assignment is optional, but if used sets the initial level of the modifier.

The FROM GROUPNAME section is optional if the modifier is also found in one of the defined MODS() modifier groups, but required if not.

Do NOT include the #REF, FROM, or assignment within any braces around names. (You may include the entire statement inside quotes or braces to separate it from other modifier blocks within the INITMODS().)

Using the Self-Control Roll modifier example from the #FORMAT section above, you can refer to it in an INITMODS() like this:

```
initmods(#ref Self-Control Roll = 2 FROM Self-Control)
```

which will look for it in the Modifiers group called Self-Control, and assign it at level 2 (12 or less).

In a trait, it might look like this:

```
[DISADVANTAGES]
<Mundane Mental>
Chronic Depression, -15, mods(Self-Control), page(B126), cat(Mundane, Mental), initmods( #ref Self-Control
Roll = 2 )
```

Note that because the mods(Self-Control) tag exists in this item, GCA will find the modifier even though we don't explicitly provide a group to look in.

You can mix with the full-definition method, too:

```
[DISADVANTAGES]
<Mundane Mental>
Chronic Depression, -15, mods(Self-Control), page(B126), cat(Mundane, Mental),
initmods( #ref Self-Control Roll = 2 | _
      Mitigator: Meds, -60%, group(_General), page(B112), mitigator(yes), shortname(w/Meds)_
      )
```

AddMods() EXPANDED

#REF is also supported in ADDMODS(). The existing behavior is preserved, but now you can also use this new format:

```
ADDMODS( #REF MODIFIERNAME [ = X ]  [ FROM GROUPNAME ] TO TARGETTRAIT )
```

This can also be mixed with the other valid ADDMODS() references (MODGROUP:MODNAME or #NEWMOD()). For example, using the Chronic Pain modifier group example from the #FORMAT directive above:

```
addmods( ( "Chronic Pain:Interval", "#ref Frequency=2 from Chronic Pain" ) to "DI:Chronic Pain" )
```

The first reference is the existing format of MODGROUP:MODNAME, and the second is the new format using #REF. Note also that the new reference format allows for specifying a level for leveled modifiers, while the old reference format does not.

Keep in mind that if you leave off the FROM portion in this usage, the MODS() tag of the TARGETTRAIT will be used to find the modifier. In our example above, we could have used

```
#ref Frequency=2
```

because the Chronic Pain disadvantage has the Chronic Pain modifier group in its MODS() tag.

Adds() EXPANDED

#REF is also supported in the WITH and AND blocks of the ADDS() tag, instead of having to include full definitions.

The existing behavior is preserved, but now you can also use this new format to reference existing modifiers:

```
ADDS( TRAIT _
      [ WITH "#REF MODIFIERNAME [ = X] [ FROM GROUPNAME ]" _ ]
      [ AND  "#REF MODIFIERNAME [ = X] [ FROM GROUPNAME ]" _ ]
)
```

For example:

```
Chronic Depression 4, -15, mods(Self-Control), page(B126), cat(Mundane, Mental),
      adds(_
            DI:Chronic Pain=3 _
                  with " #ref Interval=3 " _
                  and  #ref Frequency _
            )
```

If the FROM block is left off, the MODS() tag of the newly added trait will be used when searching for the specified modifiers. In the example above, Interval and Frequency will be found because the Chronic Pain disadvantage has those in its MODS() tag.

Creates() EXPANDED

#REF is also supported in the WITH and AND blocks of the CREATES() tag, instead of having to include full definitions. See ADDS() above for details.

Data File Commands

#ChoiceList — Expanded

Expanded the number of altXlist() up to alt9list().

Added the new tag AutoAccept() to the list of tags for #ChoiceList. If AutoAccept() has any content, then GCA will simply accept the default values for the #ChoiceList without bothering to show the dialog to the user.

TotalCost(), PicksAllowed(), and the values for Default() are now math-enabled, so can be given formulas. Since parsing within those tags is done on spaces, you should enclose the formulas in quotes or braces.

#CloneMod — New

Added #CloneMod data file command. Works like #Clone, but clones a modifier.

> #CloneMod SourceGroup:SourceModifier as NewGroup:NewName

Unlike traits, modifiers don't have an item type, but they're unique within their groups, so #CloneMod requires specifying the modifier group and the full name of the modifier, as shown.

Enclose the entire Source or New sections in quotes or braces as required.

#DeleteModGroup — New

Allows for deleting an entire modifier group and all its modifiers at once.

> #DeleteModGroup "TargetGroup"

Specify the mod group name to remove, and it is removed, along with all the modifiers it contains.

For example:

> #DeleteModGroup "Burning Attack Limitations"

#DeleteModsFromGroup — New

Allows for deleting specific modifiers from existing data.

> #DeleteModsFromGroup "TargetGroup" ModName [, ModName2 [...]]

Similar to #DeleteFromGroup in structure, you specify the modifier group from which you're deleting items, then a space, then all the modifiers from that group that you want to delete, in a comma separated list. Enclose the modifier group name in quotes, and enclose any modifier names that contain commas inside quotes as well.

For example:

> #DeleteModsFromGroup "Burning Attack Enhancements" Partial Dice, "Partial Dice, Per Die"

#Get NEW

#GET allows you to retrieve #STORE values.

#GET, and its companion #STORE, are for use in building other library features, such as SELECT() or #CHOICE dialogs. You can use them to insert what will be character features, such as storing a formula that's inserted into a trait, but remember that #GET is only processed when a trait is added to the character.

The #GET template looks like this:

```
#GET( VARNAME )
```

and simply replaces the entirety of the #GET command with the retrieved value.

A couple examples:

```
[ADS]
<_TESTING>
#Store TestData=This is a Test
Example 1 (#GET(TESTDATA)), 5/10

#Store DialogTitle=Test Dialog
#Store DialogText = Hello!~PThis is a dialog to demonstrate the #Store and #Get commands.~PTry it!
#Store DialogList = Item 1 Chosen, Item 2 Chosen, Item 3 Chosen
Example 2 (%choice%), 5/10, x(#ChoiceList(list(#get(dialoglist)),
      Title(#get(dialogtitle)),Text(#get(dialogtext)) ))
```

In Example 1, the name extension becomes **This is a Test** when added to the character.

In Example 2, a CHOICE dialog pops up when added to a character, and the dialog is populated with the #STORE values.

#IF NEW

Allows you to have conditional file processing.

The structure is like this:

```
#IF WANT = VALUE [THEN]
      [...]
[#ELSEIF WANT2 = VALUE2 [THEN]]
      [...]
[#ELSE]
      [...]
#END[IF]
```

Notice that the THEN keywords are optional; GCA will ignore them if found, but you can simply leave them out if desired. Same with the IF in #ENDIF; GCA considers any #END to be the end of the current #IF structure.

You may nest #IF..#END blocks.

Note that there is *no* expression evaluator involved here. Support exists for a tiny set of very specific comparisons, which I'll cover here. If you try anything else, GCA will consider the block FALSE and continue on, happily ignoring that section (and provide an error in the log if you have verbose book processing turned on).

The two types of WANT = VALUE comparisons currently supported are these:

1) Check for a file

```
FILELOADED = NAMEOFFILE
```

This allows you to see if a file has been loaded before this one. The file currently being processed does *not* count. You must use the exact file name that GCA has loaded (ignoring path information, and ignoring case).

GCA supports the following aliases for FILELOADED, so you may use whichever you remember: FILELOADED, LOADEDFILE, FILEISLOADED, LOADED, BOOKLOADED, BOOKISLOADED.

NAMEOFFILE may be in quotes or braces.

2) Check for a trait

```
TRAITLOADED = NAMEOFTRAIT
```

This allows you to see if a particular trait exists in the current library data. Anything loaded before this comparison is a possibly valid subject.

GCA supports the following aliases for TRAITLOADED, so you may use whichever you remember: TRAITLOADED, TRAITEXISTS, TRAITPRESENT, LOADEDTRAIT.

NAMEOFTRAIT may be in quotes or braces, and must be in the standard fully qualified format, with prefix and full name and extension, as applicable.

#MergeModTags NEW

Works like #MERGETAGS, but for modifiers.

```
#MERGEMODTAGS IN TARGETGROUP:TARGETMODIFIER WITH TAGLIST
```

Unlike traits, modifiers don't have an item type, but they're unique within their groups, so #MERGEMODTAGS requires specifying the modifier group and the full name of the modifier, as shown.

For example:

```
#mergemodtags in "Burning Attack Enhancements:Partial Dice" with "page(EX)"
```

The ALL keyword is **not** supported.

#ReplaceModTags NEW

Works like #REPLACETAGS, but for modifiers.

```
#REPLACEMODTAGS IN TARGETGROUP:TARGETMODIFIER WITH TAGLIST
```

Unlike traits, modifiers don't have an item type, but they're unique within their groups, so #REPLACEMODTAGS requires specifying the modifier group and the full name of the modifier, as shown.

For example:

```
#replacemodtags in "Burning Attack Enhancements:Partial Dice" with "cost(+1/+2),formula(%level * 5)"
```

The ALL keyword is **not** supported.

#Store NEW

#STORE is a data file directive that allows storing text as a named block for use elsewhere in the Library, such as

```
#store warning=Don't Do That!
```

These are basically global variables on the Library level, rather than at the trait level, and in concept are similar to Lists, but store only a single item of text, rather than a group of related text items.

#STORE, and its companion #GET, are for use in building other library features, such as SELECT() or #CHOICE dialogs. You can use them to insert what will be character features, such as storing a formula that's inserted into a trait, but remember that #GET is only processed when a trait is added to the character.

The #STORE template looks like this:

```
#STORE VARNAME = TEXT
```

The VARNAME should be simple, but if necessary (when it includes an = sign) it can be enclosed in quotes or braces.

The TEXT bit can be whatever text you want, so long as it conforms to GCA's data file rules (one line, or put together from multiple lines using line continuation). Don't enclose TEXT in quotes or braces unless you want them wherever the text is going to be inserted. Includes support for ~P (that's a tilde followed by a capital P) for inserting a carriage return/line feed into the text, which is converted during #GET retrieval.

Be conscious of where the text is going to go; don't include things like unbalanced parens or other characters that might result in the destination no longer being correctly parseable after the value is inserted.

Whitespace at the front or end of either part is trimmed.

See #GET for examples of use.

#Verbose NEW

This has only two options, On or Off, like so:

```
#Verbose On
#Verbose Off
```

This allows you to turn on VerboseBookProcessing for a specific file, or portion of a file, for testing purposes, without having to have full-on verbosity for every file you're loading. (Technically, this sets a different property, and does not affect the user's VerboseBookProcessing setting at all; if they have that turned on, this will have no impact on it at all, and they'll get the verbosity they desire.)

It should be considered polite to remove these directives from files before they're made publicly available, as many users will find verbose book processing quite annoying.

Special Notes

Gains Bonuses NEW

Gains bonuses are bonuses that a trait claims for itself, based on some other trait, rather than having to edit that other trait to have it give a bonus. GCA has never supported Gains bonuses, instead requiring that you create a work-around using Gives bonuses, when a Gains would have been more convenient.

Now, we have initial support for Gains bonuses, using the existing structure and the existing GIVES() tag, but with a tiny change in wording, using a FROM keyword instead of the TO keyword.

See the GIVES() tag for more information.

Modes EXPANDED

Modes have received more attention in GCA5, and have a new handler. To the user, this should be transparent, but it means that expecting mode-enabled behavior from non-mode-enabled tags may not work as expected.

Here are all the mode-enabled tags:

Acc()
ArmorDivisor()
Break()
Bulk()
CharAcc()
CharArmorDivisor()
CharBreak()
CharBulk()
CharDamage()
CharDamType()
CharEffectiveST()
CharMinST()
CharParry()
CharParryScore()
CharRadius()
CharRangeHalfDam()
CharRangeMax()
CharRcl()
CharReach()
CharRof()
CharShots()
CharSkillScore()
CharSkillUsed()
Damage()
DamageBasedOn()
DamageIsText()
DamType()
Dmg()
ItemNotes()
LC()
MinST()
MinSTBasedOn()
Mode()
Notes()
Parry()
Radius()
RangeHalfDam()
RangeMax()
Rcl()
Reach()
ReachBasedOn()
Rof()
Shots()
SkillUsed()
STCap()

Substitutions NEW

Support has been added for a Substitutions system, which allows one trait to substitute for another when GCA is tracking down references. This system applies to character traits, and does not do substitutions on a library basis.

With this system, you could create a Guns (Small) skill that substitutes for Guns (Pistol), and then when a reference to Guns (Pistol) is being sought by GCA, Guns (Small) would be considered a valid return.

You create a substitution table by using the SUBSFOR() tag in the item that is intended to substitute for the others. For example, our Guns (Small) trait would include subsfor("SK:Guns (Pistol)").

Multiple entries can be submitted at once in SUBSFOR() by separating them with commas. The trait type prefix code is required in the SUBSFOR() definition.

Implementation is currently limited, but should be working in most places, such as defaults and weapons tables. Note that the item that is the substitute also qualifies as an existing instance of anything for which it substitutes, so adding one of those items after the substitution table is created would be considered a duplicate by GCA.

User Targetable Bonuses NEW

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

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

See the GIVES() tag for more information.

Name Extension Directives NEW

GCA now supports some directives specific to name extensions, in specific circumstances.

#Any NEW

#ANY will return the first valid non-zero trait with the given name.

> #ANY [OF LIST][EXCEPT LIST]

#Best NEW

#BEST will return the best (highest non-zero).

> #BEST [OF LIST][EXCEPT LIST]

#Worst NEW

#WORST will return the worst (lowest non-zero).

> #WORST [OF LIST][EXCEPT LIST]

#None NEW

#NONE will look only at traits without a name extension.

> #NONE

Details

Each of these directives allows you to specify how you'd like to look for the trait, given possible variations of the name extension.

So, if you have several possible Flight advantages, but any one of them works just to say that you have Flight, you could use a reference like

> SK:Flight (#any)

to find any of them.

There are also optional clauses to contract the acceptable pool of traits by the name extensions found, should you wish to do that. All these directives, excluding #NONE, support the OF and EXCEPT clauses.

OF specifies the extensions that are valid, and only traits with one of those specified extensions will be considered.

EXCEPT specifies the extensions that are not valid, and only traits that don't have a specified extension will be considered.

LIST is a list of the applicable name extensions for that clause. Enclose individual list items within quotes or braces if they include a comma or one of the other keywords.

The OF and EXCEPT clauses should never have any reason to be used at the same time, but should you do so, order isn't important so long as the clauses always follow the #keyword.

#NONE doesn't support clauses because it retrieves only traits without extensions, so there's no extension to check against them.

Solver

The Solver respects these new directives, so they should be working almost any time GCA is looking up a trait.

Note that #BEST and #WORST can be tricky, and should be used cautiously, as results may not be as expected in all cases, depending on what tag or value you're looking for. Only numeric values can be judged.

The Text Function Solver also respects these directives. However, because it returns text values, there's no good way to determine best or worst, and it therefore treats all of them as #ANY, where any non-empty value satisfies the request.

Needs Checking

The Needs system also supports these directives when checking prerequisites for any traits that are properly identified with the correct prefix tags, and that have a valid base name.

GCA will assemble a list of all traits of the correct type that have the given base name, and then process them in this fashion:

#ANY - all traits will be evaluated and used to test the condition, but only until the first trait found that satisfies the given condition; or until all traits fail.

#BEST - all traits will be evaluated, and the one with the best (highest) value will be used to test the condition.

#WORST - all traits will be evaluated, and the one with the worst (lowest) value will be used to test the condition.

Obviously, this only works for numeric values and conditions that evaluate numeric expressions. Inactive traits are not valid, and won't be considered.

Here's a completely arbitrary and silly example of the NEEDS() usage:

> needs(SK:Animal Handling (#any of equines, big cats) = 12, SK:Animal Handling (#best) > 15, SK:Animal Handling (#worst except "raptors") < 8)

In this example, only the ANIMAL HANDLING (EQUINES) or ANIMAL HANDLING (BIG CATS) skills will satisfy the need for an ANIMAL HANDLING skill at 12 or higher, but any version at all will work to satisfy the need that some skill is over 15, and finally, the worst ANIMAL HANDLING skill must be less than 8, but can't be ANIMAL HANDLING (RAPTORS) because that's excluded.