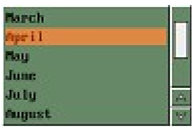
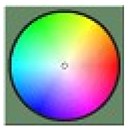
```
+---------------+---------------------------------------------------+
| ###### ###### |    .--. .         .-.                             |
| ##  ## ##   # |    |   )|        (   ) o                          |
| ##  ##  ##    |    |--' |--. .-.  `-.  .  .-...--.--. .-.         |
| ######  ##    |    |  \ |  |(   )(   ) | (   ||  |  |(   )        |
| ##      ##    |    '   `'  `-`-'  `-'-' `-`-`|'  '  `-`-'`-       |
| ##     ##   # |                           ._.'                    |
| ##     ###### | Sources & Documents placed under the MIT License. |
+---------------+---------------------------------------------------+
```

Table Of Contents





```
+-----------------------------------------------------------------+
| Done by RhoSigma, R.Heyder, provided AS IS, use at your own risk. |
| Find me in the QB64 Forum or mail to support@rhosigma-cw.net for  |
| any questions or suggestions. Thanx for your interest in my work. |
+-----------------------------------------------------------------+
```

The GuiTools Framework

by RhoSigma, Roland Heyder

INTRODUCTION

Long time ago, as my interest for Computers and programming them did arise, my first one was the Commodore Amiga 500, six years later came the A4000 Tower (back then called the Flagship of the Amiga family). Both are still fully functional and at least once a week up and running. During the years I've learned to program them in Assembler, C/C++ and the A4000T in ARexx (integrated part of the AmigaOS 2.0+) and of course in various BASIC dialects available for the Amiga. Well, no need to say that still using them todays is more a fun factor, than for any serious work. However, the fact they are still functional was and is an immense contribution to this project.

While the programming of GUIs was always possible using "Intuition", the AmigaOS's windowing subsystem, it was often a pain in the ass. With release 2.0 of the AmigaOS in 1990, a new subsystem called "GadTools" was introduced, which does wrap a lot of the detail work into some more handy functions and so it made the task of creating GUIs much easier. Read more about GadTools here, if you are interested in it's concepts. Obviously here comes the first idea for my project from, what i mean is the name **GadTools** -> **GuiTools** :)

Now, starting with a similar name, the goal is to make a similar GUI toolkit, or re-implementing GadTools for QB64, of course with a certain amount of artistic (better "programmers") freedom. One of the things i do like most about GadTools, is its simple yet elegant design. Everything is linear or rectangular and just needs a handfull of color pens to get drawn nicely to the screen. Another feature is the fact, that the entire gadget handling and updating stuff is done internally. Ie. if i did define a "Button", then i just need to listen for a so called "GADGETUP" message to know when it was clicked, but how this click happens (ie. knowing if the mousepointer is currently over that "Button" and if the left mouse button was pressed in just this moment, as well as visualizing the click by temporarily changing the imagery of the "Button"), is not in my concern as application programmer, the GUI system will take this part. Finally, as GUIs were mostly statically hardcoded into the project under the early versions of AmigaOS, with GadTools now it's possible to add, change and remove GUI objects at any time during runtime easily, thanks to its open OOP like approach.

However, a thing what GadTools is lacking, is a layout engine for automatic recalculation of all object positions and sizes when the font is changed or when the window is resized by the user. Also keyboard shortcuts can be assigned to objects, but GadTools does not warn, if the same shortcut is used for multiple objects. For my project GuiTools, I'll take over these two limitations as features. First it makes things easier to implement, 2nd the application developer is forced to double check his work, hence preventing many mistakes in the first run. Especially the layout should be made clear and logic already by the application developer, ie. optic grouping and/or separation of objects using frames and rulers, prevent overlapping of objects, make sure text fields and buttons are big enough to fit for its text, and most important if pagers/tabs are used, that child objects are placed inside the actual area of its parent page/tab, not somewhere outside.

INTRODUCTION (continued)

Another important thing to mention is, that GadTools (and my project GuiTools too) does provide gadgets and menus, which are created in a window or screen, but it does not provide the windows/screens itself, as this is handled by "Intuition" in the AmigaOS, not by GadTools. In different words, i do not try to reinvent "MS Windows" with its Desktop and Taskbar, unlike a lot of other GUIs do (see GuiReviews), but i build my GUI on top of the regular QB64 program window.

My first try on the GUI is a couple years ago, but it came to an end very quick, as i made the mistake (in my opinion) to use UDTs for object storage. To keep track of objects an array of UDTs was used. Problem was, every object type needs some different properties, but an UDT array can only store one type of UDT. Method one, using a big complex UDT, which covers the properties of any object type, but just filling the needed fields for each object type leaving other fields blank. This had been an immense waste of memory, so method two was using one UDT per object type and collect them in several arrays with interconnections between them for related objects. This had been made the entire handling and implementation overcomplicated, hence my motivation went down and the project was on ice. What i did need was a nice method to give every object exactly the needed properties (at best also allowing the app developer to add optional arbitrary data without the need to went into the object code making changes to handle it), while still being able to store everything in one array.

Well, late September 2015 i was once more working with my Amiga 4000T and stumbled over another nice feature of the AmigaOS again, which never came to my thoughts because of its obvious use in almost any part of the Amiga Operating System. It's so called "Tag Lists", a kind of associated arrays with key/value pairs, which can also be passed as varargs array to functions in C/C++ (see TagLists). The immediate thought was, that's it!! - Of course, i had to rethink its implementation in QB64. To avoid the need to reimplement a lot of AmigaOS's tag list functions, it was logic to do it in strings rather than arrays in QB64, because we have no heuristic functions to easily compare or search through arrays, but we have those for strings.
The **Tag Strings** were born.

After a review of some GadTools/TagLists related documentation i did start the implementation and testing of my **Tag String API**. While doing this i already got new ideas for the GuiTools project every minute and was quickly sure, that this will become much more than a GUI library, but a whole concept framework for GUI application development with QB64. To make it complete i did first scan through all my sources and cut out any commonly useful routine i could find and tossed them together in one file.

The remaining task was to set a few goals and rules to get not lost in the middle of the way, but my motivation was restored and the project was ready to pickup some pace again.

The Goals

LOW REQUIREMENTS

Use of **QB64 SDL 0.954** keywords only, and only limited **DECLARE LIBRARY** usage for some low level stuff:

- stick to 256 color screens, provided with a good palette which will give neat results for a wide range of images using Floyd-Steinberg dithering
- simple AmigaOS3/GadTools like design with just some minor enhancements borrowed from the beautiful MUI (Magic User Interface)
- not just hardcoded GUI design, but also possible interactive/dynamic GUI object manipulation during runtime
- updating and drawing GUI objects is kept internal to objects and called handlers, don't bother app developer with that stuff

```
- no layout checks   \  (basically means, the app developer
- no shortcut checks /   knows, what he's doing)
```

FUTUREPROVE CODE

OOP like program design, as far as possible in QB64, which allows for easy class and feature additions in the future:

- Object Class = **FUNCTION**,
- Object Method = local **GOSUB**s within that functions
- all functionality is coded on contract base, ie. if you give correct input, then GuiTools will guarantee the correct (expected) results
- thanks to **Tag Strings** all classes (**FUNCTION**s) can use the same common argument list and no changes in any class's argument lists is required when adding new features to a class in the future, as newly required parameters can simply be added as new tags, hence no "*wrong number of arguments*" errors appear in the IDE due to a changed argument list when upgrading to a newer version, simply exchange the set of GuiTools include files and then recompile your projects, that's all
- Tag Strings also allow easily for any kind and number of arbitrary user data directly associated with every object, as any unknown tags are simply ignored by the object classes, so they will not hurt the class's operation
- a general input handler which returns event based messages, app developers just need to take actions according to the received events, such as mousemove, keystroke, focus, click etc.
- all object setup and communication is done using Tag Strings, ie. every method is fed with tags, also the results (if any) are returned as tags

DEVELOPER SUPPORT

Every class shall provide support functions to simplify recurring tasks:

- remapping functions for named **OBJECT** tags
- event comparing functions for object identification

Providing my Notepad++ user defined language definition containing all actually used keywords and settings. Note this is an extra download, which you can find in the "Bonus Stuff" on the same Forum page as GuiTools.

The Tag String API

INTRODUCTION

As the GuiTools Framework's Object Classes commonly operate on Tag Strings, hence using them as inputs and results, it's a good idea to read thru this chapter first to get an overview of the **Tag String API**'s functions.

Always use these routines to create and manipulate any tags and/or tag strings, never write any tags or Tag Strings as literals, even if you know the used introducer/separator and terminator tokens. Follow this rule even if it's hard, seems to be overkill in some situations and does require a lot more writing, it will ensure the integrity of the tag API and will make future changes less painful.

Note that many of the following **SUB**s and **FUNCTION**s will have a side effect on the given **tagString$** argument. It's done so by intention. Note also, that the **tagName$** is mandatory where used, otherwise the whole thing wouldn't make any sense at all. In fact, be prepared for any kind of misbehavior, if you fail to provide a **tagName$**. The names have no case and are whitespace proof, ie. "ThisName" = "THISNAME" = "This NAME" = " T   h I s   n Am E   ".

INTERNALS

Before i come to the function reference, let me write a few words for better understanding of the Tag Strings concept. A tag is very similar to a simple **variable = value** assignment, whereas **variable is the tagName$** argument and **value the tagData$** argument in the following reference. Further a Tag String (**tagString$** in the reference) is a concatenation of many single tags.

To be able to uniquely identify a single tag and its data within the tag string some special tokens are required for tag and data separation. These tokens should never be used within tag names and/or data to preserve its uniqueness as special tokens within the entire tag string. You can change the tokens in the include file TagSupport.bi, if required.

Within a tag string all single tags are in general <u>typeless</u>, hence anything could be stored in it, doesn't matter if string, integer or floating point value. How each tag is interpreted is up to the respective implementation. The GuiTools Object Classes will therefor define the types of the used tags in its documentation.

SUBs & FUNCTIONs

(see TagSupport.bm)

Numbers in square brackets designate the Framework's version in which the subroutine, function or tag first appeared, hence the minimum version of GuiTools needed, if you wanna use that subroutine, function or tag.

SUB AddTag (tagString$, tagName$, tagData$) [v0.1]
Add a new tag to the tag string. Note this routine is <u>internal</u>, you should always use SetTag instead to avoid multiple tags of the same name get added to the tag string.

SUBs & FUNCTIONs (continued)
(see TagSupport.bm)

SUB SetTag (tagString$, tagName$, newData$) [v0.1]
Set data of the given tag in the tag string. Will add a new tag, if the named tag does not exist yet in the tag string.

**FUNCTION NewTag$** (tagName$, tagData$) [v0.1]
Create a new tag. Similar to AddTag, but this will just return the new tag instead of adding it to any tag string.

SUB RemTag (tagString$, tagName$) [v0.1]
Search given tag and remove it from the tag string. Will do nothing, if the tag does not exist in the tag string.

SUB RemTags (tagString$, remNames$) [v0.1]
Same as RemTag, but used to remove many tags according to the given (comma separated) list of tag names. Can remove a single tag too, although RemTag would be more efficient in that case.

**FUNCTION GetTag$** (tagString$, tagName$) [v0.1]
Search given tag in the tag string and return it. Will return empty, if the tag does not exist in the tag string. Note that a found tag is not checked for validity, it's returned as it is in the tag string.

**FUNCTION GetTags$** (tagString$, getNames$) [v0.1]
Same as GetTag$, but used to get many tags according to the given (comma separated) list of tag names. Can return a single tag too, although GetTag$ would be more efficient in that case.

**FUNCTION ExtractTag$** (tagString$, tagName$) [v0.1]
Search given tag in the tag string, return it and also remove it from the tag string. Will return empty and remove nothing, if the tag does not exist in the tag string. Note that a found tag is not checked for validity, it's returned as it was in the tag string before removal.
Used in conjunction with GetTagName$ to process (unknown) user tags.

**FUNCTION GetTagName$** (tagString$) [v0.1]
Look for the first tag in the tag string and return its name. Will return empty, if no tags exist in the tag string. Used in conjunction with ExtractTag$ to process (unknown) user tags.

**FUNCTION GetTagData$** (tagString$, tagName$, defData$) [v0.1]
Search given tag in the tag string and return its associated data. If the tag does not exist or appears to be invalid, then return the provided default data instead.

FUNCTION UpdateTag% (oldTagString$, tagName$, newTagString$) [v0.5]
Search given tag in both, new and old Tag Strings and update its data from new to old string, if the data is different or the named tag does not even exist yet in the old string. Will return true, if the old string had to be updated, or false, if the tag its data were either equal in both (old/new) Tag Strings or the given tag did not even exist in the new tag string.

SUBs & FUNCTIONs (continued)

(see TagSupport.bm)

FUNCTION UpdateTags% (oldTagString$, updNames$, newTagString$) [v0.5]
Same as UpdateTag%, but used to update many tags according to the given (comma separated) list of tag names. This function will return true, if at least one of the given tags had to be updated, otherwise it's false.
Can update a single tag too, although UpdateTag% would be more efficient in that case.

FUNCTION ValidateTags% (tagString$, chkNames$, remInvalid%) [v0.1]
Will return a boolean value (0/-1) according to the check whether all given tags (comma separated list of tag names) exist in the tag string and also have valid (non-empty) data. May also remove any invalid tags upon request (**remInvalid%** = true (non-zero)). Will always return true, if no tag names to check are specified (**chkNames$** = empty).

FUNCTION BoolTagTrue% (tagString$, tagName$) [v0.1]
Will return a boolean value (0/-1) according to the check whether the given boolean tag in the tag string is true (tag found and data is numeric non-zero or string "true"/"yes"/"on") or false (tag not found or data is numeric zero or not string "true"/"yes"/"on").

FUNCTION ToggleBoolTag% (tagString$, tagName$) [v0.1]
Toggle the state of the given boolean tag in the tag string and return a boolean value (0/-1) according to the tag its new state.

IMPLEMENTATION RULES

Tags may be either public or private. It's up to the respective class or function documentation, whether private tags are described or not. You should never mess with undocumented tags or make any assumptions about its usage. In general they are probably undocumented for a good reason.

Tag names should be unique within the given **tagString$** argument, hence you should avoid multiple tags of the same name within the same **tagString$** (use SetTag instead of AddTag for automatic replacement of same named tags). Be aware, that always the first found tag is used/returned, if multiples exist.

Boolean tags can be either true or false, which can be achieved by using the following values for those tags:

```
  true  => string "true", "yes" or "on" or numeric non-zero (best -1)
  false => string "false", "no" or "off" or numeric zero (0)
```

These values can be given as literals, variables or defined CONSTs, but note that even numerics must be given/defined as strings (eg. "-1"), as all tags in fact are strings. The case of the strings doesn't matter.

Boolean tags should be considered false, if omitted. The provided functions BoolTagTrue% and ToggleBoolTag% will support this assumption. However, the function ValidateTags% will not, as it don't know which of the tags it has to validate are boolean tags. This should regularly not be a problem, as a function which depends on boolean tags, will probably use one of the first two mentioned functions anyways to validate these tags.

IMPLEMENTATION RULES (continued)

You may also add any kind and number of arbitrary user tags to any method call. This means, you can store any data within any object, such as special IDs, counters, comments, debug information etc., whatever you may need for your application. You can give your tags at **INIT** time, add/change them via the **SET** method and later retrieve them using **GET**, just as it's done with the regular tags. However, to not interfere with the regular GuiTools tags, you must give your own tags any kind of underscore separated ID such as a name prefix/suffix (eg. **USER_DEBUG** or **MY_ID** etc.).

Every object class shall only check for the tags which it needs for correct operation, hence all additional tags in the given **tagString$** (inclusive your own tags) must be ignored for the internal class operations. So we don't need to be worry to have any unused/optional or user tags in any object's **tagString$**, it won't hurt the class or its correct function.

The Common Tags

OBJECT HANDLE & ERRORS

NewTag$("OBJECT", "0") (Long) [v0.1]
This is the object handle (index) which is returned by the INIT method on success and which is required by all other method calls to designate the desired object on which the method shall be called. You regularly don't need to add this tag via the syntax shown above, as it is usually ready to use within the returned handle, so it is sufficient to specify the desired **handle$** for any additional method calls and just add the tags required by the called method respectively (Xxx = any object class):

```
 eg. handle$ = XxxC$("INIT", initTags$)
 ie. handle$ = XxxC$("INIT", NewTag$(...) + NewTag$(...) etc.)

   then any further method call on this object would look like:

 eg. result$ = XxxC$("METHOD", handle$ + methodTags$)
 ie. result$ = XxxC$("METHOD", handle$ + NewTag$(...) + NewTag$(...) etc.)
```

Note that the OBJECT tag always refers to an object of the called class. If any additional other object types are required for any method call, then those tags will be named directly (eg. PAGEROBJECT, IMAGEOBJECT etc.) and will usually be added by using OBJECT tag remapping functions provided by the respective object classes (eg. PagerTag$(), ImageTag$() etc.).

I have coded all classes/methods on a contract base, ie. if you provide all required and correct input, then the class/method will guarantee the expected result. By this means you only need to check for the next two tags during development of your own applications to find out faulty behaviors. Once you are ready to release your work, no more errors in your GUI object base should exist and any checks for the following tags can be omitted.

NewTag$("ERROR", "error message") (String) [v0.1]
An error message, may be returned by any method call and means the method could not finish properly. The most common reasons for this are omitting required tags for any method call, using any unknown method names or by passing a wrong object type to a specific class. At least the latter one may easily be avoided by calling methods (except the INIT method) using the generic class GenC$, which will route the call to the correct class depending on the given object handle.

NewTag$("WARNING", "warning message") (String) [v0.2]
A warning message, may be returned by any method call and means the method could still finish by using internal fallbacks, or by simply ignoring the faulty condition if it does not vitally affect the internal operations. However, especially for the DRAW method this may mean, that an object is not rendered the way it was intended.

To properly check for the existence of any ERROR/WARNING tags the Tag String **FUNCTION** ValidateTags% should be used on the result of any method call, see also **FUNCTION** ShowErr$ in file **GuiApp.bas** (respectively YourAppName.bas).

OBJECT POSITIONS & SIZES

As these tags usually designate screen pixel positions and/or sizes, it is needless to say that its values should be in general all positives. Some exceptions exist to allow relative positioning according to another object, but these will be listed in the appropriate places.

NewTag$("LEFT", "0") (Integer) [v0.1]
The left pixel position where to place the new object.

NewTag$("TOP", "0") (Integer) [v0.1]
The top pixel position where to place the new object.

NewTag$("WIDTH", "0") (Integer) [v0.1]
The pixel width of the new object.

NewTag$("HEIGHT", "0") (Integer) [v0.1]
The pixel width of the new object.

NewTag$("LENGTH", "0") (Integer) [v0.1]
The pixel length of the new object, if it's only one dimension.

OBJECT NAMES (LABEL)

For better understanding, labels are the descriptive texts placed outside right next to any objects.

NewTag$("LABEL", "descriptive text") (String) [v0.1]
Any descriptive text for the new object.

NewTag$("LABELHIGH", "true") (Boolean) [v0.1]
If set true, then the **guiHighPen%** is used to print the label, otherwise it will default to **guiShinePen%**. The default pens are specified in the include file **GuiClasses.bi**, but may be overwritten by user preferences.

NewTag$("LABELPLACE", "placement") (String) [v0.1]
Label placement relative to the new object. Is one of the choices "left", "above", "right" or "below". If this tag is empty or omitted, then it will default to "above".

Object's label position calculation is optimized for use of the QB64 inbuilt fonts 8/14/16. Using custom fonts (_LOADFONT) may cause unprecise positions, you may use the next two tags for manual adjustment.

NewTag$("LABELMOVEX", "0") (Integer) [v0.1]
Delta X adjustment in pixels for the label, may be negative.

NewTag$("LABELMOVEY", "0") (Integer) [v0.1]
Delta Y adjustment in pixels for the label, may be negative.

OBJECT CONTENT (TEXT)

For better understanding, text means the text content placed inside objects, such as the text output in TextC$ objects or texts placed onto Rulers/Frames.

NewTag$("TEXT", "text") (String) [v0.1]
Any text content for the new/actual object.

NewTag$("TEXTPLACE", "placement") (String) [v0.1]
Text placement relative to the object. It works similar to LABELPLACE, but has different options depending on the object type, which will be listed in the appropriate places.

Object's text position calculation is optimized for use of the QB64 inbuilt fonts 8/14/16. Using custom fonts (_LOADFONT) may cause unprecise positions, you may use the next two tags for manual adjustment. Another use could be to make some space for an embedded image or symbol.

NewTag$("TEXTMOVEX", "0") (Integer) [v0.1]
Delta X adjustment in pixels for the text, may be negative.

NewTag$("TEXTMOVEY", "0") (Integer) [v0.1]
Delta Y adjustment in pixels for the text, may be negative.

OBJECT ASSIGNMENTS

Note well, that all object assignment tags are checked for validity and fitness for its purpose before they are actually assigned to any object. In fact, be prepared for ERROR/WARNING returns of the respective method calls.

The multiple choice object types of GuiTools will require that the choices or options are given as a predefined list. The following tag is used to assign such a list to those objects. Note that the given list object is not allowed for public access in your application while assigned and will ERROR out, but usually you can query all tags (incl. user tags) of the actually chosen list record directly via the GET method of the parent multiple choice object.

NewTag$("LISTOBJECT", "0") (Long) [v0.1]
The index of an already initialized ListC$ object, which shall be assigned to the new/actual object. In alternative you may simply use the list its **handle$** from the respective INIT call and use the tag remapping support call ListTag$(listHandle$) instead (see ListClass.bm).

Since GuiTools v0.2 some, since v0.5 all operational objects (Button, Slider, Textfield, Cycler etc.) do accept an image and/or symbol assigned to it. How much sense this makes for the one or other object type I'll leave up to your own imagination, the handling logic for it is there at least. Use the next two tags to assign the image/symbol to any object.

NewTag$("IMAGEOBJECT", "0") (Long) [v0.2]
The index of an already initialized ImageC$ object, which shall be assigned to the new/actual object. In alternative you may simply use the image its **handle$** from the respective INIT call and use the tag remapping support call ImageTag$(imageHandle$) instead (see ImageClass.bm).

OBJECT ASSIGNMENTS (continued)

NewTag$("SYMBOLOBJECT", "0") (Long) [v0.2]
The index of an already initialized **SymbolC$** object, which shall be assigned to the new/actual object. In alternative you may simply use the symbol its **handle$** from the respective **INIT** call and use the tag remapping support call **SymbolTag$**(symbolHandle$) instead (see **SymbolClass.bm**).

GuiTools supports the use of pagers/tabs. To keep track of whether an object is visible or invisible depending on the active pager/tab, every object needs to know to which pager/tab it belongs to. The following tag will assign any new object to the specified pager/tab.

NewTag$("PAGEROBJECT", "0") (Long) [v0.1]
The index of an already initialized **PagerC$** object, to which the new object shall belong to. In alternative you may simply use the pager its **handle$** from the respective **INIT** call and use the tag remapping support call **PagerTag$**(pagerHandle$) instead (see **PagerClass.bm**). Note that the object association is for the internal refresh logic only, you are still responsible to place the new object entirely within the pagers display area by using the appropriate screen coordinates for the LEFT, TOP, WIDTH and HEIGHT tags.

OBJECT CONTROL

Here are three control tags which are known by most operational objects such as Buttons, Checkboxes, Cycler etc.. There are some more control tags, but they are only used for certain object types, hence they are listed in the appropriate places.

NewTag$("TOOLTIP", "info text") (String) [v0.8]
Use this tag to add a short help or info text for the object, which is displayed, if the mousepointer does rest over the object for a short time. If required, then you may write multiple text lines separated with a vertical bar "|". Check your given tooltips to make sure they fit into the available display area.

NewTag$("SHORTCUT", **MakeShortcut$**(...)) (String) [v0.1] [v0.5]
This tag defines a keyboard shortcut for the object. This was simply a single char in early versions of GuiTools. Since v0.5 this must be the return value of the support **FUNCTION** **MakeShortcut$** (see **GuiClasses.bm**). The used shortcut sequence is also added to any given TOOLTIP for the object or will be shown alone, if no tooltip is defined.

NewTag$("DISABLED", "true") (Boolean) [v0.1]
If set true, then the object is drawn overlayed by a pixel grid and is not selectable until it is enabled again, which would be done by **SET** this tag its value to "false".

The GuiTools Object Classes

INTRODUCTION

The Object Classes do provide the access to the available GUI objects like Buttons, Sliders, Checkboxes etc., every GUI object type is represented as one class which does encapsulate the entire functionality for that object type in so called "Methods". These methods are the only way supported for you (the User) to alterate or query the object's properties. Of course all objects are listed in a big array, which elements could be manipulated by everybody, but don't blame me if you get in trouble by doing so. The classes are featuring a common and easy to use interface to access all required functions of any GUI objects, so please don't play stunts with the array.

Each class is implemented as single **FUNCTION** with embedded **GOSUB** subroutines which represent its methods. If you call a method of any class, hence making a call to the respective **FUNCTION**, then first a method dispatcher is entered which designates the required and optional tags for the called method and then makes a **GOSUB** call into the method's subroutine. These subroutines will then verify the validity of the given object (except for the **INIT** method) and the given tags. If everything looks valid, then the method will be executed and after that **RETURN** to the dispatcher, which will then do **EXIT FUNCTION**.

COMMON USER METHODS

Every class supports four general methods for object initialization, object property alteration, to query object properties and finally to dispose no longer needed objects. Note that all methods will return Tag Strings, hence you get informations back the same way you pass it in (Xxx = any class):

handle$** = **XxxC$("**INIT**", initTags$) [v0.1]
This method will initialize a new object of the respective class. It will return the object handle on success (**OBJECT** tag), but may also return a **WARNING** or **ERROR** tag, if something went wrong. This method does always trigger a GUI refresh event to make sure that the new object is rendered immediately (except if it belongs to an currently invisible pager/tab).

When initializing your GUI, then make sure to place your objects in a logic and sensible way. GuiTools does not have a layout engine and does no error checking on your layout. So first avoid overlapping of objects and second if you use pagers/tabs, then keep in mind that there should be no objects on the main screen area which is covered by the pagers/tabs. Instead of that assign the objects to the respective pager/tab by using the **PAGEROBJECT** tag as seen in section **Object Assignments** in the previous chapter.

result$ = **XxxC$**("**SET**", **handle$** + setTags$) [v0.1]
This method will alterate the given object's (**handle$**) properties taking the new values from the given tags, if required this method will also redraw the object. Note that some properties may not allowed to be changed after object initialization, such as position, sizes, shortcuts & labels. If given nevertheless, then those properties are simply ignored. Be aware of **WARNING** or **ERROR** tag returns here either.

COMMON USER METHODS (continued)

result$ = XxxC$("GET", **handle$** + NewTag$("TAGNAMES", "...")) [v0.1]
Use this method to query any of the given object's (**handle$**) properties according to the given comma separated list of tags (TAGNAMES tag). This may be required after the object was changed (eg. an input field) and you need to retrieve the input for the further program flow. As said above, the values are returned as regular tags. Not existing properties will be simply ignored, but there may be other WARNING or ERROR conditions.

result$ = XxxC$("KILL", **handle$**) [v0.1]
This method will delete the given object (**handle$**). Note that the handle becomes invalid after calling this method and you better should clear it out (handle$ = "") after this method call, if there is a chance that it will be reused in your program. This method does always trigger a GUI refresh event to make sure that the disposed object is removed from the display immediately. Be aware of WARNING or ERROR tag returns.
Note that deleting an PagerC$ class object will also delete <u>all</u> objects which are placed on that particular pager/tab. Also deleting a ListC$ class object will first delete <u>all</u> records stored in that list.

Note that some classes may provide even more methods to fulfill specific tasks. Those methods will be listed in the respective class documentations.

COMMON INTERNAL METHODS

The next two methods are also known to most classes. Note that this methods are just mentioned here to make the common methods list complete, you usually <u>never</u> need to call these methods directly, as drawing and updating of the objects is handled internally within the classes.

result$ = XxxC$("DRAW", **handle$** + drawTags$) [v0.1]
This internal method will render the given object (**handle$**) according to its internal known data. It is called whenever the object is known to be currently visible and needs to be redrawn due to changed data/states.

result$ = XxxC$("UPDATE", **handle$** + updateTags$) [v0.1]
This internal method is responsible to update the properties of the given object (**handle$**) according to the given input events. It's called whenever new inputs from keyboard and/or mouse are available.

The List Class

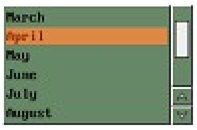
NAME

```
  ListC$ -- Operate doubly linked list objects for data storage [v0.1]
            (see ListClass.bm)
```

DESCRIPTION

OPERATION

The Listview Class



NAME

ListviewC$ -- Operate vertically scrollable list objects [v0.7]
(see ListviewClass.bm)

DESCRIPTION

The Listview Class does provide all necessary stuff to easily build and operate vertically scrollable lists. You should already be familiar with the **Tag String API** and the **Common User Methods** to understand the class descriptions.

You simply define a rectangular area for the entire listview object. The options (list entries) must be stored in a ListC$ object and assigned to the listview via the **LISTOBJECT** tag at **INIT** time. The listview object will then automatically arrange the entries and a scroller within the given rectangle by taking your desired line spacing, the current text font and your maybe given images or symbols for each entry into account.

Any option related images/symbols must be directly added to the respective entries in the **List object** and should have a 4:3 aspect ratio with a height close or equal to the used text font height. However, using the **SPACING** tag you could make enough space for larger images/symbols or such with another aspect ratio, but how this would look in the listview is written on another piece of paper.

OPERATION

The user may ...

METHODS & PROPERTIES (copy from method dispatcher)

Each method supports a number of tags to specify the object's properties and behavior. These are passed in through the **tagString$** argument of the class. There may be required and/or optional tags. Calling methods is an easy task, if you already know about **Object Handles** and the **Common User Methods**.

```
  CASE "Init"
      tagsReq$ = "LEFT,TOP,WIDTH,HEIGHT,LISTOBJECT,"
      tagsOpt$ = "PAGEROBJECT,IMAGEOBJECT,SYMBOLOBJECT,SPACING,ACTUAL," +_
                 "LABEL,LABELHIGH,LABELPLACE,LABELMOVEX,LABELMOVEY," +_
                 "IMAGEFILE,IMAGEHANDLE,AREA," +_
                 "TOOLTIP,SHORTCUT,READONLY,DISABLED," '+ user tags
      'NOTE: If ACTUAL is not given or outside list bounds, then it
      '      will default to one (1st list entry/record), may also be
      '      zero for no list entry preselection. Entry SPACING will
      '      default to one (1), if omitted. The spacing is a minimum
      '      value here and may be slightly raised internally to better
      '      arrange the entries in the visible Listview area.
      '      If READONLY is true, then ACTUAL and SHORTCUT is ignored.
      GOSUB meInit
      'Result Tags: ERROR/WARNING or OBJECT
  CASE "Set"
      tagsReq$ = "OBJECT,"
      tagsOpt$ = "IMAGEOBJECT,SYMBOLOBJECT,LISTOBJECT," +_
                 "ACTUAL,READONLY,DISABLED," '+ user tags
      'NOTE: If ACTUAL is not given or outside list bounds, then it
      '      will default to one (1st list entry/record), may also be
      '      zero for no list entry preselection.
      GOSUB meSet
      'Result Tags: ERROR/WARNING or empty
  CASE "Get"
      tagsReq$ = "OBJECT,"
      tagsOpt$ = "TAGNAMES," 'list may include internal + user tags
      'NOTE: The ACTUAL tag holds the index of the actually chosen option
      '      according to the order in the currently assigned multiple
      '      choice option list, and you may also directly query any tags
      '      of the respective list record here (DATA + user tags).
      GOSUB meGet
      'Result Tags: ERROR/WARNING or requested tags (may be empty)
  CASE "Kill"
      tagsReq$ = "OBJECT,"
      tagsOpt$ = ""
      GOSUB meKill
      'Result Tags: ERROR/WARNING or empty
```

USED COMMON TAGS

(see also **KnownTags.txt**)

- **Object Positions & Sizes** (**LEFT,TOP,WIDTH,HEIGHT**)
- **Object Label** (**LABEL,LABELHIGH,LABELPLACE,LABELMOVEX,LABELMOVEY**)
- **Object Assignment** (**LISTOBJECT,PAGEROBJECT,IMAGEOBJECT,SYMBOLOBJECT**)
- **Object Control** (**TOOLTIP,SHORTCUT,DISABLED**)
- user tags as mentioned under the tag **Implementation Rules**

The tags **IMAGEFILE**, **IMAGEHANDLE** and **AREA** are working according to its image class descriptions and may be used to directly designate a background image for the newly created object.

The Slider Class

NAME

SliderC$ -- Operate horizontal and vertical slider objects [v0.4]
           (see SliderClass.bm)

DESCRIPTION

The Slider Class does provide all necessary stuff to easily build and operate horizontal or vertical movable sliders. You should already be familiar with the **Tag String API** and the **Common User Methods** to understand the class descriptions.

You do simply choose the desired alignment by specifying the aspect ratio via the WIDTH and HEIGHT tags. If WIDTH > HEIGHT, then the alignment is horizontal and vise versa. Of course you could also specify a square size, but the sliding knob wouldn't be movable in that case, so it makes no sense.

Each slider will have a minimum and maximum level (integers only). The initial level may be set at init time or will default to the minimum.

OPERATION

The user may adjust the current slider level by grabbing the knob with the mousepointer and move it to a new position. The knob will stick to the mouse as long as the left mousebutton is held down, even if the pointer runs out of focus while moving. But be aware, that the new level position is only taken, if the mousebutton is released while in focus or max. 50 pixels around the slider (extended focus). If the release happens out of this area, then the knob will jump back to its old position. The level indicator (either the internal or an external via ModelC$ Forwarder interconnection) will update continuously while moving the knob.

The user may also click into the free space before and behind the knob or use the mousewheel to decrement or increment the slider level in steps of one. Holding down the Shift modifier key when using the mousewheel will accelerate the level counting by factor ten.

Note that mousewheel movements are always routed to the focused object or the last object, which had focus, if currently no one has focus. Hence, once the slider had focus, the user can control it via the wheel until he hits another object with the mousepointer.

METHODS & PROPERTIES (copy from method dispatcher)

Each method supports a number of tags to specify the object's properties and behavior. These are passed in through the **tagString$** argument of the class. There may be required and/or optional tags. Calling methods is an easy task, if you already know about **Object Handles** and the **Common User Methods**.

```
  CASE "Init"
      tagsReq$ = "LEFT,TOP,WIDTH,HEIGHT,MINIMUM,MAXIMUM,"
      tagsOpt$ = "PAGEROBJECT,IMAGEOBJECT,SYMBOLOBJECT," +_
                 "LABEL,LABELHIGH,LABELPLACE,LABELMOVEX,LABELMOVEY," +_
                 "LEVEL,ALTMIN,ALTMAX,TEXTMOVEX,TEXTMOVEY,NOSHOW," +_
                 "IMAGEFILE,IMAGEHANDLE,AREA,TOOLTIP,DISABLED," '+ user tags
      'NOTE: If given, LEVEL will be clipped to either MINIMUM or MAXIMUM
      '      when out of valid range. Default LEVEL = MINIMUM.
      '      TEXTMOVEX/Y can be used to adjust the level output within
      '      the slider's knob (unless NOSHOW is set true).
      GOSUB meInit
      'Result Tags: ERROR/WARNING or OBJECT
  CASE "Set"
      tagsReq$ = "OBJECT,"
      tagsOpt$ = "IMAGEOBJECT,SYMBOLOBJECT,MINIMUM,MAXIMUM," +_
                 "LEVEL,TEXTMOVEX,TEXTMOVEY,NOSHOW,DISABLED," '+ user tags
      'NOTE: If given, LEVEL will be clipped to either MINIMUM or MAXIMUM
      '      when out of valid range.
      GOSUB meSet
      'Result Tags: ERROR/WARNING or empty
  CASE "Get"
      tagsReq$ = "OBJECT,"
      tagsOpt$ = "TAGNAMES," 'list may include internal + user tags
      GOSUB meGet
      'Result Tags: ERROR/WARNING or requested tags (may be empty)
  CASE "Kill"
      tagsReq$ = "OBJECT,"
      tagsOpt$ = ""
      GOSUB meKill
      'Result Tags: ERROR/WARNING or empty
```

USED COMMON TAGS

(see also **KnownTags.txt**)

- **Object Positions & Sizes** (**LEFT,TOP,WIDTH,HEIGHT**)
- **Object Label** (**LABEL,LABELHIGH,LABELPLACE,LABELMOVEX,LABELMOVEY**)
- **Object Text** (**TEXTMOVEX,TEXTMOVEY**)
- **Object Assignment** (**PAGEROBJECT,IMAGEOBJECT,SYMBOLOBJECT**)
- **Object Control** (**TOOLTIP,DISABLED**)
- user tags as mentioned under the tag **Implementation Rules**

The tags **IMAGEFILE**, **IMAGEHANDLE** and **AREA** are working according to its image class descriptions and may be used to directly designate a background image for the newly created object.

OBJECT SPECIFIC TAGS

NewTag$("MINIMUM", "0") (Long) [v0.4]
The minimum level the new slider can represent. This is the level, if the knob is in its leftmost or bottom-most position on a horizontal or vertical slider respectively. May be negative.

NewTag$("MAXIMUM", "0") (Long) [v0.4]
The maximum level the new slider can represent. This is the level, if the knob is in its rightmost or topmost position on a horizontal or vertical slider respectively. May be negative.

NewTag$("LEVEL", "0") (Long) [v0.4]
The initial level the new slider's knob shall be adjusted to. Note that the given value is clipped to either MINIMUM or MAXIMUM, if out of range. If this tag is omitted, then the level will default to the MINIMUM.

NewTag$("ALTMIN", "text") (String) [v0.8]
This tag will define an alternative level output, if a slider is on its minimum level (eg. a volume slider with a range 0-10 could alternatively display "Mute" when reaching level zero). Note that this does only change the output, the LEVEL tag does still hold the numeric value.

NewTag$("ALTMAX", "text") (String) [v0.8]
This works similar to the tag above, but is used to define an alternative output for the slider's maximum level.

NewTag$("NOSHOW", "0") (Boolean) [v0.4]
Normally the current level of the slider's knob is printed within the knob's shape. However, depending on the sliding range and the pixel size of the slider the printed level may not fit into the knob, which then looks ugly. In such cases you may set this tag true to suppress the level output. Instead you can place eg. a TextC$ object in an appropriate place next to the slider and establish a interconnection between its TEXT tag and the LEVEL tag of this slider using an ModelC$ Forwarder object. The primary interconnection object/tag should be the slider with its LEVEL.

This approach is also suitable to easily implement a slider with a range of floating point numbers rather than integers. Just use a ModelC$ Ratio object instead of a Forwarder object, which will divide the slider's current LEVEL by 10, 100, 1000 etc.. Then, in your application you would not use the slider's LEVEL directly to retrieve its value, but you would read eg. the TEXT tag of the connected TextC$ object instead, which holds the divided (and maybe formatted) floating point value. For this approach the slider with its LEVEL tag should be given as the multiplier object to the ModelC$ interconnection.

Eg. A slider range 0-300 plus interconnection Ratio = 100 would result
    in a floating point range 0.00-3.00 in the connected object.

The Scroller Class

NAME

ScrollerC$ -- Operate horizontal and vertical scroller objects [v0.5]
(see ScrollerClass.bm)

DESCRIPTION

The Scroller Class does provide all necessary stuff to easily build and operate horizontal or vertical movable scrollers. You should already be familiar with the **Tag String API** and the **Common User Methods** to understand the class descriptions.

You do simply choose the desired alignment by specifying the aspect ratio via the **WIDTH** and **HEIGHT** tags. If **WIDTH** > **HEIGHT**, then the alignment is horizontal and vise versa. To get a well shaped and usable scroller you should not specify a square size, also the short:long side aspect ratio must be at least 1:4 to make sure there's enough space to move the scroller. Each scroller will also have two arrow buttons for single stepping.

OPERATION

The user may adjust the current scroller position by grabbing the knob with the mousepointer and move it to a new position. The knob will stick to the mouse as long as the left mousebutton is held down, even if the pointer runs out of focus while moving. But be aware, that the new position is only taken, if the mousebutton is released while in focus or max. 50 pixels around the scroller (extended focus). If the release happens out of this area, then the knob will jump back to its old position.

The user may also click into the free space before and behind the knob or use the mousewheel in combination with a pressed Ctrl modifier key to decrement or increment the scroller in steps of one visible page. The arrow buttons or the mousewheel without any modifier key can be used to decrement or increment in steps of one line or column. If an arrow button is pressed for more than 0.5 seconds, then the line/column step will be repeated until the button is released. Also, holding down the Shift modifier key when using the mousewheel will accelerate the line/column step by factor ten.

Note that mousewheel movements are always routed to the focused object or the last object, which had focus, if currently no one has focus. Hence, once the scroller had focus, the user can control it via the wheel until he hits another object with the mousepointer.

METHODS & PROPERTIES (copy from method dispatcher)

Each method supports a number of tags to specify the object's properties and behavior. These are passed in through the **tagString$** argument of the class. There may be required and/or optional tags. Calling methods is an easy task, if you already know about **Object Handles** and the **Common User Methods**.

```
  CASE "Init"
      tagsReq$ = "LEFT,TOP,WIDTH,HEIGHT,VISIBLENUM,"
      tagsOpt$ = "PAGEROBJECT,IMAGEOBJECT,SYMBOLOBJECT,TOTALNUM,TOPNUM," +_
                 "IMAGEFILE,IMAGEHANDLE,AREA,TOOLTIP,DISABLED," '+ user tags
      'NOTE: If not given, then TOTALNUM = VISIBLENUM, any given TOPNUM
      '      will be clipped to be in valid range. Default = 0.
      GOSUB meInit
      'Result Tags: ERROR/WARNING or OBJECT
  CASE "Set"
      tagsReq$ = "OBJECT,"
      tagsOpt$ = "IMAGEOBJECT,SYMBOLOBJECT," +_
                 "VISIBLENUM,TOTALNUM,TOPNUM,DISABLED," '+ user tags
      'NOTE: If given, TOPNUM will be clipped to be in valid range.
      GOSUB meSet
      'Result Tags: ERROR/WARNING or empty
  CASE "Get"
      tagsReq$ = "OBJECT,"
      tagsOpt$ = "TAGNAMES," 'list may include internal + user tags
      GOSUB meGet
      'Result Tags: ERROR/WARNING or requested tags (may be empty)
  CASE "Kill"
      tagsReq$ = "OBJECT,"
      tagsOpt$ = ""
      GOSUB meKill
      'Result Tags: ERROR/WARNING or empty
```

USED COMMON TAGS

(see also **KnownTags.txt**)

- **Object Positions & Sizes** (**LEFT,TOP,WIDTH,HEIGHT**)
- **Object Assignment** (**PAGEROBJECT,IMAGEOBJECT,SYMBOLOBJECT**)
- **Object Control** (**TOOLTIP,DISABLED**)
- user tags as mentioned under the tag **Implementation Rules**

The tags **IMAGEFILE**, **IMAGEHANDLE** and **AREA** are working according to its image class descriptions and may be used to directly designate a background image for the newly created object.

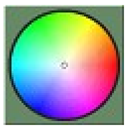
OBJECT SPECIFIC TAGS

NewTag$("**VISIBLENUM**", "0") (Integer) [v0.5]
The scroller always needs to know the number of visible lines or columns of the text view (or pixel height/width of the image view) it is connected to. This number is represented by the relative scroll knob size compared to the scroller's container size. It shouldn't be zero.

NewTag$("**TOTALNUM**", "0") (Long) [v0.5]
This is the current total number of lines or columns in the text buffer (or height/width of the entire image), which this scroller is connected to. This number is projected to the scroller's available container size. If this tag is omitted, then it will default to **VISIBLENUM** (entire text or image is visible), hence the knob would fill the entire container and wouldn't be movable.

NewTag$("**TOPNUM**", "0") (Long) [v0.5]
The first line or column number (or pixel position) which is displayed in the visible area. If required, then this value is clipped to be in the valid range. This value is represented by the current scroller knob position within the scroller's container. Note that this counter starts at zero, which is also the default, if this tag is omitted.

The Colorwheel Class



NAME

ColorwheelC$ -- Operate HSB color picker wheel objects [v0.4]
(see ColorwheelClass.bm)

DESCRIPTION

The Colorwheel Class does provide all necessary stuff to easily build and operate full hue/saturation range color picker wheels. You should already be familiar with the **Tag String API** and the **Common User Methods** to understand the class descriptions.

The hue is represented on the wheel's arc, while the saturation is on the wheel's radius. You should always combine a colorwheel with a **Slider Object** for brightness selection to allow the user to easily adjust all three color components of the HSB model.

The **FUNCTION**s **HSB32~&** and/or **HSBA32~&** (**ConvertSupport.bm**) can be used to convert the picked HSB values into a 32-bit ARGB color value.

OPERATION

The user may adjust the current hue and saturation by grabbing the knob with the mousepointer and move it to a new position. The knob will stick to the mouse as long as the left mousebutton is held down, even if the pointer runs out of focus while moving. But be aware, that the new position is only taken, if the mousebutton is released while in focus in the square area surrounding the colorwheel. If the release happens out of this area, then the knob will jump back to its old position.

The user may also click into the free space around the knob or use the mousewheel to change the values. Using the mousewheel without pressing any modifier keys will adjust the saturation, while pressing the Ctrl modifier does switch the mousewheel to change the hue instead. Also, additionally holding down the Shift modifier key will accelerate the step by factor ten.

Note that mousewheel movements are always routed to the focused object or the last object, which had focus, if currently no one has focus. Hence, once the colorwheel had focus, the user can control it via the mousewheel until he hits another object with the mousepointer.

METHODS & PROPERTIES (copy from method dispatcher)

Each method supports a number of tags to specify the object's properties and behavior. These are passed in through the **tagString$** argument of the class. There may be required and/or optional tags. Calling methods is an easy task, if you already know about **Object Handles** and the **Common User Methods**.

```
CASE "Init"
    tagsReq$ = "LEFT,TOP,WIDTH,"
    tagsOpt$ = "PAGEROBJECT,IMAGEOBJECT,SYMBOLOBJECT,HUE,SATURATION," +_
               "IMAGEFILE,IMAGEHANDLE,AREA,TOOLTIP,DISABLED," '+ user tags
    'NOTE: Default HUE and SATURATION is zero, if not given. Also HUE
    '      is clipped to a range of 0-360 (degrees), SATURATION will
    '      be in range 0-100 (percent). HEIGHT is always = WIDTH.
    GOSUB meInit
    'Result Tags: ERROR/WARNING or OBJECT
CASE "Set"
    tagsReq$ = "OBJECT,"
    tagsOpt$ = "IMAGEOBJECT,SYMBOLOBJECT,HUE,SATURATION," +_
               "DISABLED," '+ user tags
    GOSUB meSet
    'Result Tags: ERROR/WARNING or empty
CASE "Get"
    tagsReq$ = "OBJECT,"
    tagsOpt$ = "TAGNAMES," 'list may include internal + user tags
    GOSUB meGet
    'Result Tags: ERROR/WARNING or requested tags (may be empty)
CASE "Kill"
    tagsReq$ = "OBJECT,"
    tagsOpt$ = ""
    GOSUB meKill
    'Result Tags: ERROR/WARNING or empty
```

USED COMMON TAGS

(see also **KnownTags.txt**)

- **Object Positions & Sizes** (**LEFT,TOP,WIDTH**)
- **Object Assignment** (**PAGEROBJECT,IMAGEOBJECT,SYMBOLOBJECT**)
- **Object Control** (**TOOLTIP,DISABLED**)
- user tags as mentioned under the tag **Implementation Rules**

The tags **IMAGEFILE**, **IMAGEHANDLE** and **AREA** are working according to its image class descriptions and may be used to directly designate a background image for the newly created object.

OBJECT SPECIFIC TAGS

NewTag$("**HUE**", "0") (Integer) [v0.4]
The current hue value represented by the colorwheel's knob position.
If omitted, it will default to zero, otherwise it will be clipped into the valid range from 0-360 degrees. The color gradient starts with red (0°), goes over yellow (60°) to green (120°), then further over cyan (180°) to blue (240°) and finally over magenta (300°) back to red (360° or 0°).

NewTag$("**SATURATION**", "0") (Integer) [v0.4]
The current saturation represented by the colorwheel's knob position.
If omitted, it will default to zero, otherwise it will be clipped into the valid range from 0-100 percent. As lower this value is, as more the color turns into a gray value.