

Libraries_Manual

COLLABORATORS

	<i>TITLE :</i> Libraries_Manual		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY		July 18, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1	Libraries_Manual	1
1.1	Amiga® RKM Libraries: 15 GadTools Library	1
1.2	15 GadTools Library / Elements of GadTools	1
1.3	15 / Elements of GadTools / GadTools Tags	2
1.4	15 GadTools Library / GadTools Menus	2
1.5	15 / GadTools Menus / The NewMenu Structure	4
1.6	15 / GadTools Menus / Functions for GadTools Menus	6
1.7	15 // Functions for GadTools Menus / Creating Menus	7
1.8	15 // Functions for GadTools Menus / Layout of the Menus	8
1.9	15 // Functions for GadTools Menus / Layout for Individual Menus	9
1.10	15 // Functions for GadTools Menus / Freeing Menus	9
1.11	15 / GadTools Menus / GadTools Menus and IntuiMessages	10
1.12	15 / GadTools Menus / Restrictions on GadTools Menus	10
1.13	15 / GadTools Menus / Language-Sensitive Menus	10
1.14	15 GadTools Library / GadTools Gadgets	11
1.15	15 / GadTools Gadgets / The NewGadget Structure	12
1.16	15 / GadTools Gadgets / Creating Gadgets	13
1.17	15 / GadTools Gadgets / Handling Gadget Messages	14
1.18	15 / GadTools Gadgets / IDCMP Flags	16
1.19	15 / GadTools Gadgets / Freeing Gadgets	16
1.20	15 / GadTools Gadgets / Modifying Gadgets	17
1.21	15 / GadTools Gadgets / The Kinds of GadTools Gadgets	18
1.22	15 // The Kinds of GadTools Gadgets / Button Gadgets	19
1.23	15 // Kinds of GadTools Gadgets / Text-Entry and Number-Entry Gadgets	19
1.24	15 // The Kinds of GadTools Gadgets / Checkbox Gadgets	22
1.25	15 // The Kinds of GadTools Gadgets / Mutually-Exclusive Gadgets	22
1.26	15 // The Kinds of GadTools Gadgets / Cycle Gadgets	23
1.27	15 // The Kinds of GadTools Gadgets / Slider Gadgets	24
1.28	15 // The Kinds of GadTools Gadgets / Scroller Gadgets	26
1.29	15 // The Kinds of GadTools Gadgets / Listview Gadgets	28

1.30	15 // The Kinds of GadTools Gadgets / Palette Gadgets	30
1.31	15 // Kinds of GadTools / Text-Display and Numeric-Display Gadgets	31
1.32	15 // The Kinds of GadTools Gadgets / Generic Gadgets	32
1.33	15 / GadTools / Functions for Setting Up GadTools Menus and Gadgets	32
1.34	15 /// GetVisualInfo() and FreeVisualInfo()	32
1.35	15 // Setting Up GadTools Menus and Gadgets / CreateContext()	34
1.36	15 / GadTools Gadgets / Creating Gadget Lists	34
1.37	15 / GadTools Gadgets / Gadget Refresh Functions	36
1.38	15 / GadTools Gadgets / Other GadTools Functions	37
1.39	15 // Other Functions / GT_FilterIMsg() and GT_PostFilterIMsg()	37
1.40	15 // Other GadTools Functions / DrawBevelBox()	39
1.41	15 / GadTools Gadgets / Gadget Keyboard Equivalents	39
1.42	15 // Keyboard Equivalents / Denoting a Gadget's Keyboard Equivalent	40
1.43	15 /// Implementing a Gadget's Keyboard Equivalent Behavior	40
1.44	15 / GadTools Gadgets / Restrictions on GadTools Gadgets	42
1.45	15 / GadTools Gadgets / Documented Side-Effects	42
1.46	15 GadTools Library / Function Reference	43

Chapter 1

Libraries_Manual

1.1 Amiga® RKM Libraries: 15 GadTools Library

GadTools is a new library in Release 2 that is designed to simplify the task of creating user interfaces with Intuition. GadTools offers a flexible and varied selection of gadgets and menus to help programmers through what used to be a difficult chore.

Intuition, the Amiga's graphical user interface, is a powerful and flexible environment. It allows a software designer a great degree of flexibility in creating dynamic and powerful user interfaces. However, the drawback of this flexibility is that programming even straightforward user interfaces can be complicated, and certainly difficult for first-time Intuition programmers.

What the Gadget Toolkit (GadTools) attempts to do is harness the power of Intuition by providing easy-to-use, high-level chunks of user interface. GadTools doesn't pretend to answer all possible user interface needs of every application but by meeting the user interface needs of most applications, GadTools greatly simplifies the problem of designing user-friendly software on the Amiga. (For applications with special needs, custom solutions can be created with Intuition's already-familiar gadgets or its new Boopsi object-oriented custom gadget system; GadTools is compatible with these.)

Elements of GadTools	GadTools Gadgets
GadTools Menus	Function Reference

1.2 15 GadTools Library / Elements of GadTools

GadTools is the easy way to program gadgets and menus. With GadTools, the system handles the detail work required to control gadgets and menus so the application uses less code and simpler data structures.

Another key benefit of GadTools is its standardized and elegant look. All applications that use GadTools will share a similar appearance and behavior. Users will appreciate a sense of instant familiarity even the first time they use a product.

GadTools provides a significant degree of visual consistency across multiple applications that use it. For instance, in Release 2, the Preferences editors, the Workbench "Information" window and Commodities Exchange share the same polished look and feel thanks to GadTools. There is also internal consistency between different elements of GadTools; the look is clean and orderly. Depth is used not just for visual embellishment, but as an important cue. For instance, the user is free to select symbols that appear inside a "raised" area, but "recessed" areas are informational only, and clicking in them has no effect.

GadTools is not amenable to creative post-processing or hacking by programmers looking to achieve a result other than what GadTools currently offers. Software developers whose needs extend beyond the standard features of GadTools should create custom gadgets that share the look and feel of GadTools by using either BOOPSI or by directly programming gadgets at a lower level. See the chapters on "Intuition Gadgets" and "BOOPSI" for more information. Follow the GadTools rules. Only in this way may GadTools grow and improve without hindrance, even allowing new features to automatically appear in future software when reasonable.

GadTools Tags

1.3 15 / Elements of GadTools / GadTools Tags

Many of the GadTools functions use TagItem arrays or tag lists to pass information across the function interface. These tag-based functions come in two types, one that takes a pointer to an array of tag items and one that takes a variable number of tag item arguments directly in the function call. In general, the second form, often called the varargs form because the call takes a variable number of arguments, is provided for convenience and is internally converted to the first form. When looking through the Autodocs or other Amiga reference material, the documentation for both forms is usually available in the array-based function description.

All GadTools tags begin with a leading "GT". In general, they also have a two-letter mnemonic for the kind of gadget in question. For example, slider gadgets recognize tags such as "GTSL_Level". The GadTools tags are defined in <libraries/gadtools.h>. Certain GadTools gadgets also recognize other Intuition tags such as GA_Disabled and PGA_Freedom, which can be found in <intuition/gadgetclass.h>.

For more information on tags and tag-based functions, be sure to see the "Utility Library" chapter in this manual.

1.4 15 GadTools Library / GadTools Menus

GadTools menus are easy to use. Armed only with access to a VisualInfo data structure, GadTools allows the application to easily create, layout and delete Intuition menus.

Normally, the greatest difficulty in creating menus is that a large number of structures must be filled out and linked. This is bothersome since much of the required information is orderly and is easier to do algorithmically than to do manually. GadTools handles this for you.

There are also many complexities in creating a sensible layout for menus. This includes some mechanical items such as handling various font sizes, automatic columnization of menus that are too tall and accounting for space for checkmarks and Amiga-key equivalents. There are also aesthetic considerations, such as how much spacing to provide, where sub-menus should be placed and so on.

GadTools menu functions support all the features that most applications will need. These include:

- * An easily constructed and legible description of the menus.
- * Font-sensitive layout.
- * Support for menus and sub-menus.
- * Sub-menu indicators (a ">>" symbol attached to items with sub-menus).
- * Separator bars for sectioning menus.
- * Command-key equivalents.
- * Checkmarked and mutually exclusive checkmarked menu items.
- * Graphical menu items.

With GadTools, it takes only one structure, the NewMenu structure, to specify the whole menu bar. For instance, here is how a typical menu strip containing two menus might be specified:

```
struct NewMenu mynewmenu[] =
{
    { NM_TITLE, "Project",      0 , 0, 0, 0,},
    { NM_ITEM, "Open...",      "O", 0, 0, 0,},
    { NM_ITEM, "Save",          "S", 0, 0, 0,},
    { NM_ITEM, NM_BARLABEL,     0 , 0, 0, 0,},
    { NM_ITEM, "Print",         0 , 0, 0, 0,},
    { NM_SUB, "Draft",          0 , 0, 0, 0,},
    { NM_SUB, "NLQ",            0 , 0, 0, 0,},
    { NM_ITEM, NM_BARLABEL,     0 , 0, 0, 0,},
    { NM_ITEM, "Quit...",       "Q", 0, 0, 0,},

    { NM_TITLE, "Edit",         0 , 0, 0, 0,},
    { NM_ITEM, "Cut",            "X", 0, 0, 0,},
    { NM_ITEM, "Copy",           "C", 0, 0, 0,},
    { NM_ITEM, "Paste",          "V", 0, 0, 0,},
    { NM_ITEM, NM_BARLABEL,     0 , 0, 0, 0,},
    { NM_ITEM, "Undo",           "Z", 0, 0, 0,},

    { NM_END, NULL,             0 , 0, 0, 0,},
};
```

This NewMenu specification would produce the two menus below:

Figure 15-1: Two Example Menus

The NewMenu arrays are designed to be read easily. The elements in the NewMenu array appear in the same order as they will appear on-screen. Unlike the lower-level menu structures described in the "Intuition Menus" chapter earlier, there is no need to specify sub-menus first, then the menu items with their sub-menus, and finally the menu headers with their menu items. The indentation shown above also helps highlight the relationship between menus, menu items and sub-items.

The NewMenu Structure	GadTools Menus And IntuiMessages
GadTools Menus Example	Restrictions on GadTools Menus
Functions for GadTools Menus	Language-Sensitive Menus

1.5 15 / GadTools Menus / The NewMenu Structure

The NewMenu structure used to specify GadTools menus is defined in <libraries/gadtools.h> as follows:

```
struct NewMenu
{
    UBYTE nm_Type;
    STRPTR nm_Label;
    STRPTR nm_CommKey;
    UWORD nm_Flags;
    LONG nm_MutualExclude;
    APTR nm_UserData;
};
```

nm_Type

The first field, nm_Type, defines what this particular NewMenu describes. The defined types provide an unambiguous and convenient representation of the application's menus.

NM_TITLE

Used to signify a textual menu heading. Each NM_TITLE signifies the start of a new menu within the menu strip.

NM_ITEM or IM_ITEM

Used to signify a textual (NM_ITEM) or graphical (IM_ITEM) menu item. Each NM_ITEM or IM_ITEM becomes a menu item in the current menu.

NM_SUB or IM_SUB

Used to signify a textual (NM_SUB) or graphical (IM_SUB) menu sub-item. All the consecutive NM_SUBs and IM_SUBs that follow a menu item (NM_ITEM or IM_ITEM) compose that item's sub-menu. A subsequent NM_ITEM or IM_ITEM would indicate the start of the next item in the original menu, while a subsequent NM_TITLE would begin the next menu.

NM_END

Used to signify the end of the NewMenu structure array. The last element of the array must have NM_END as its type.

nm_Label

NM_TITLE, NM_ITEM and NM_SUB are used for textual menu headers, menu items and sub-items respectively, in which case nm_Label points to the string to be used. This string is not copied, but rather a pointer to it is kept. Therefore the string must remain valid for the active life of the menu.

Menus don't have to use text, GadTools also supports graphical menu items and sub-items (graphical menu headers are not possible since they are not supported by Intuition). Simply use IM_ITEM and IM_SUB instead and point nm_Label at a valid Image structure. The Image structure can contain just about any graphic image (see the chapter on "Intuition Images, Line Drawing and Text" for more on this).

Sometimes it is a good idea to put a separator between sets of menu items or sub-items. The application may want to separate drastic menu items such as "Quit" or "Delete" from more mundane ones. Another good idea is to group related checkmarked items by using separator bars.

NM_BARLABEL

GadTools will provide a separator bar if the special constant NM_BARLABEL is supplied for the nm_Label field of an NM_ITEM or NM_SUB.

nm_CommKey

A single character string used as the Amiga-key equivalent for the menu item or sub-item.

Menu headers cannot have command keys. Note that assigning a command-key equivalent to a menu item that has sub-items is meaningless and should be avoided.

The nm_CommKey field is a pointer to a string and not a character itself. This was done in part because routines to support different languages typically return strings, not characters. The first character of the string is actually copied into the resulting MenuItem structure.

nm_Flags

The nm_Flags field of the NewMenu structure corresponds roughly to the Flags field of the Intuition's lower-level Menu and MenuItem structures.

For programmer convenience the sense of the Intuition MENUENABLED and ITEMENABLED flags are inverted. When using GadTools, menus, menu items and sub-items are enabled by default.

NM_MENUDISABLED

To specify a disabled menu, set the NM_MENUDISABLED flag in this field.

NM_ITEMDISABLED

To disable an item or sub-item, set the `NM_ITEMDISABLED` flag.

The Intuition flag bits `COMMSEQ` (indication of a command-key equivalent), `ITEMTEXT` (indication of a textual or graphical item) and `HIGHFLAGS` (method of highlighting) will be automatically set depending on other attributes of the menus. Do not set these values in `nm_Flags`.

The `nm_Flags` field is also used to specify checkmarked menu items. To get a checkmark that the user can toggle, set the `CHECKIT` and `MENUTOGGLE` flags in the `nm_Flags` field. Also set the `CHECKED` flag if the item or sub-item is to start in the checked state.

`nm_MutualExclude`

For specifying mutual exclusion of checkmarked items. All the items or sub-items that are part of a mutually exclusive set should have the `CHECKIT` flag set.

This field is a bit-wise representation of the items (or sub-items), in the same menu or sub-menu, that are excluded by this item (or sub-item). In the simple case of mutual exclusion, where each choice excludes all others, set `nm_MutualExclude` to $\sim(1 \ll \text{item number})$ or ~ 1 , ~ 2 , ~ 4 , ~ 8 , etc. Separator bars count as items and should be included in the position calculation. See the "Intuition Menus" chapter for more details on menu mutual exclusion.

`nm_UserData`

The `NewMenu` structure also has a user data field. This data is stored with the Intuition Menu or `MenuItem` structures that GadTools creates. Use the macros `GTMENU_USERDATA(menu)` and `GTMENUITEM_USERDATA(menuitem)` defined in `<libraries/gadtools.h>` to extract or change the user data fields of menus and menu items, respectively.

The application may place index numbers in this field and perform a switch statement on them, instead of using the Intuition menu numbers. The advantage of this is that the numbers chosen remain valid even if the menus are rearranged, while the Intuition menu numbers would change when the menus are rearranged.

Alternately, an efficient technique for menu handling is to create a handler function for each menu item and put a pointer to that function in the corresponding item's `UserData` field. When the program receives a `IDCMP_MENUPICK` message it may call the selected item's function through this field.

1.6 15 / GadTools Menus / Functions for GadTools Menus

In this section the basic GadTools menu functions are presented. See the listing above for an example of how to use these functions.

Creating Menus	Layout for Individual Menus
Layout of the Menus	Freeing Menus

1.7 15 // Functions for GadTools Menus / Creating Menus

The `CreateMenus()` function takes an array of `NewMenu` and creates a set of initialized and linked Intuition Menu, `MenuItem`, `Image` and `IntuiText` structures, that need only to be formatted before being used. Like the other tag-based functions, there is a `CreateMenusA()` call that takes a pointer to an array of `TagItems` and a `CreateMenus()` version that expects to find its tags on the stack.

```
struct Menu *CreateMenusA( struct NewMenu *newmenu,
                          struct TagItem *taglist );
struct Menu *CreateMenus( struct NewMenu *newmenu, Tag tag1, ... );
```

The first argument to these functions, `newmenu`, is a pointer to an array of `NewMenu` structures as described earlier. The tag arguments can be any of the following items:

`GTMN_FrontPen` (ULONG)

The pen number to use for menu text and separator bars. The default is zero.

`GTMN_FullMenu` (BOOL)

(New for V37, ignored under V36). This tag instructs `CreateMenus()` to fail if the supplied `NewMenu` structure does not describe a complete Menu structure. This is useful if the application does not have direct control over the `NewMenu` description, for example if it has user-configurable menus. The default is FALSE.

`GTMN_SecondaryError` (ULONG *)

(New for V37, ignored under V36). This tag allows `CreateMenus()` to return some secondary error codes. Supply a pointer to a NULL-initialized ULONG, which will receive an appropriate error code as follows:

`GTMENU_INVALID`

Invalid menu specification. For instance, a sub-item directly following a menu-title or an incomplete menu. `CreateMenus()` failed in this case, returning NULL.

`GTMENU_NOMEM`

Failed for lack of memory. `CreateMenus()` returned NULL.

`GTMENU_TRIMMED`

The number of menus, items or sub-items exceeded the maximum number allowed so the menu was trimmed. In this case, `CreateMenus()` does not fail but returns a pointer to the trimmed Menu structure.

NULL

If no error was detected.

`CreateMenus()` returns a pointer to the first Menu structure created, while all the `MenuItem` structures and any other Menu structures are attached through the appropriate pointers. If the `NewMenu` structure begins with an entry of type `NM_ITEM` or `IM_ITEM`, then `CreateMenus()` will return a pointer to the first `MenuItem` created, since there will be no first Menu

structure. If the creation fails, usually due to a lack of memory, `CreateMenus()` will return `NULL`.

Starting with V37, GadTools will not create any menus, menu items or sub-items in excess of the maximum number allowed by Intuition. Up to 31 menus may be defined, each menu with up to 63 items, each item with up to 31 sub-items. See the "Intuition Menus" chapter for more information on menus and their limitations. If the `NewMenu` array describes a menu that is too big, `CreateMenus()` will return a trimmed version. `GTMN_SecondaryError` can be used to learn when this happens.

Menus need to be added to the window with Intuition's `SetMenuStrip()` function. Before doing this, they must be formatted with a call to `LayoutMenus()`.

1.8 15 // Functions for GadTools Menus / Layout of the Menus

The `Menu` and `MenuItem` structures returned by `CreateMenus()` contain no size or positional information. This information is added in a separate layout step, using `LayoutMenus()`. As with the other tag-based functions, the program may call either `LayoutMenus()` or `LayoutMenusA()`.

```

    BOOL LayoutMenusA( struct Menu *firstmenu, APTR vi,
                      struct TagItem *taglist );
    BOOL LayoutMenus( struct Menu *firstmenu, APTR vi, Tag tag1, ... );

```

Set `firstmenu` to a pointer to a `Menu` structure returned by a previous call to `CreateMenus()`. The `vi` argument is a `VisualInfo` handle obtained from `GetVisualInfo()`. See the documentation of GadTools gadgets below for more about this call. For the tag arguments, `tag1` or `taglist`, `LayoutMenus()` recognizes a single tag:

`GTMN_TextAttr`

A pointer to an openable font (`TextAttr` structure) to be used for the menu item and sub-item text. The default is to use the screen's font.

`LayoutMenus()` fills in all the size, font and position information for the menu strip. `LayoutMenus()` returns `TRUE` if successful and `FALSE` if it fails. The usual reason for failure is that the font supplied cannot be opened.

`LayoutMenus()` takes care of calculating the width, height and position of each individual menu item and sub-item, as well as the positioning of all menus and sub-menus. In the event that a menu would be too tall for the screen, it is broken up into multiple columns. Additionally, whole menus may be shifted left from their normal position to ensure that they fit on screen. If a large menu is combined with a large font, it is possible, even with columnization and shifting, to create a menu too big for the screen. GadTools does not currently trim off excess menus, items or sub-items, but relies on Intuition to clip menus at the edges of the screen.

It is perfectly acceptable to change the menu layout by calling `ClearMenuStrip()` to remove the menus, then `LayoutMenus()` to make the change and then `SetMenuStrip()` to display the new layout. Do this when

changing the menu's font (this can be handled by a tag to `LayoutMenus()`), or when updating the menu's text (to a different language, for instance). Run-time language switching in menus will be discussed later.

1.9 15 // Functions for GadTools Menus / Layout for Individual Menus

`LayoutMenuItems()` performs the same function as `LayoutMenus()`, but only affects the menu items and sub-items of a single menu instead of the whole menu strip. Ordinarily, there is no need to call this function after having called `LayoutMenus()`. This function is useful for adding menu items to an extensible menu, such as the Workbench "Tools" menu.

For example, a single `MenuItem` can be created by calling `CreateMenus()` with a two-entry `NewMenu` array whose first entry is of type `NM_ITEM` and whose second is of type `NM_END`. The menu strip may then be removed and this new item linked to the end of an extensible menu by placing its address in the `NextItem` field of the last `MenuItem` in the menu. `LayoutMenuItems()` can then be used to recalculate the layout of just the items in the extensible menu and, finally, the menu strip can be reattached to the window.

```

    BOOL LayoutMenuItemsA( struct MenuItem *firstitem, APTR vi,
                          struct TagItem *taglist );
    BOOL LayoutMenuItems( struct MenuItem *firstitem, APTR vi,
                          Tag tagl, ... );

```

Set `firstitem` to a pointer to the first `MenuItem` in the linked list of `MenuItems` that make up the Menu. (See the "Intuition Menus" chapter for more about these structures.) Set `vi` to the address of a `VisualInfo` handle obtained from `GetVisualInfo()`. The tag arguments, `tagl` or `taglist`, may be set as follows:

GTMN_TextAttr

A pointer to an openable font (`TextAttr` structure) to be used for the menu item and sub-item text. The default is to use the screen's font.

GTMN_Menu

Use this tag to provide a pointer to the Menu structure whose `FirstItem` is passed as the first parameter to this function. This tag should always be used.

`LayoutMenuItems()` returns `TRUE` if it succeeds and `FALSE` otherwise.

1.10 15 // Functions for GadTools Menus / Freeing Menus

The `FreeMenus()` function frees all the memory allocated by the corresponding call to `CreateMenus()`.

```

    void FreeMenus( struct Menu *menu );

```

Its one argument is the `Menu` or `MenuItem` pointer that was returned by `CreateMenus()`. It is safe to call `FreeMenus()` with a `NULL` parameter, the

function will then return immediately.

1.11 15 / GadTools Menus / GadTools Menus and IntuiMessages

If the window uses GadTools menus and GadTools gadgets, then use the `GT_GetIMsg()` and `GT_ReplyIMsg()` functions described below (or `GT_FilterIMsg()` and `GT_PostFilterIMsg()`, if applicable). However, if the window has GadTools menus, but no GadTools gadgets, it is acceptable to use `GetMsg()` and `ReplyMsg()` in the usual manner.

Additionally, no context need be created with `CreateContext()` if no GadTools gadgets are used. For more about these functions, see the section on "Other GadTools Functions" later in this chapter.

1.12 15 / GadTools Menus / Restrictions on GadTools Menus

GadTools menus are regular Intuition menus. Once the menus have been laid out, the program may do anything with them, including attaching them or removing them from windows, enabling or disabling items, checking or unchecking checkmarked menu items, etc. See the documentation for `SetMenuStrip()`, `ClearMenuStrip()`, `ResetMenuStrip()`, `OnMenu()` and `OffMenu()` in the "Intuition Menus" chapter for full details.

If a GadTools-created menu strip is not currently attached to any window, the program may change the text in the menu headers (`Menu->MenuName`), the command-key equivalents (`MenuItem->Command`) or the text or imagery of menu items and sub-items, which can be reached as:

```
((struct IntuiText *)MenuItem->ItemFill)->IText
or
((struct Image *)MenuItem->ItemFill)
```

The application may also link in or unlink menus, menu items or sub-items. However, do not add sub-items to a menu item that was not created with sub-items and do not remove all the sub-items from an item that was created with some.

Any of these changes may be made, provided the program subsequently calls `LayoutMenus()` or `LayoutMenuItems()` as appropriate. Then, reattach the menu strip using `SetMenuStrip()`.

Some of these manipulations require walking the menu strip using the usual Intuition-specified linkages. Beginning with the first `Menu` structure, simply follow its `FirstItem` pointer to get to the first `MenuItem`. The `MenuItem->SubItem` pointer will lead to the sub-menus. `MenuItems` are connected via the `MenuItem->NextItem` field. Successive menus are linked together with the `Menu->NextMenu` pointer. Again, see the chapter "Intuition Menus" for details.

1.13 15 / GadTools Menus / Language-Sensitive Menus

Allowing the application to switch the language displayed in the menus, can be done quite easily. Simply detach the menu strip and replace the strings in the IntuiText structures as described above. It may be convenient to store some kind of index number in the Menu and MenuItem UserData which can be used to retrieve the appropriate string for the desired language. After all the strings have been installed, call LayoutMenus() and SetMenuStrip().

If the application has the localized strings when the menus are being created, it simply places the pointers to the strings and command shortcuts into the appropriate fields of the NewMenu structure. The menus may then be processed in the normal way.

1.14 15 GadTools Library / GadTools Gadgets

The heart of GadTools is in its ability to easily create and manipulate a sophisticated and varied array of gadgets. GadTools supports the following kinds of gadgets:

Table 15-1: Standard Gadget Types Supported by the GadTools Library

Gadget Type	Description or Example Usage
Button	Familiar action gadgets, such as "OK" or "Cancel".
String	For text entry.
Integer	For numeric entry.
Checkboxes	For on/off items.
Mutually exclusive	Radio buttons, select one choice among several.
Cycle	Multiple-choice, pick one of a small number of choices.
Sliders	To indicate a level within a range.
Scrollers	To indicate a position in a list or area.
Listviews	Scrolling lists of text.
Palette	Color selection.
Text-display	Read-only text.
Numeric-display	Read-only numbers.

GadTools gadget handling consists of a body of routines to create, manage and delete any of the 12 kinds of standard gadgets listed in table 15-1, such as buttons, sliders, mutually exclusive buttons and scrolling lists.

To illustrate the flexibility, power and simplicity that GadTools offers, consider the GadTools slider gadget. This gadget is used to indicate and control the level of something, for example volume, speed or color intensity. Without GadTools, applications have to deal directly with Intuition proportional and their arcane variables, such as HorizBody to control the slider knob's size and HorizPot to control the knob's position. Using the GadTools slider allows direct specification of the minimum and maximum levels of the slider, as well as its current level. For example, a color slider might have a minimum level of 0, a maximum level of 15 and a current level of 11.

To simplify event-processing for the slider, GadTools only sends the application a message when the knob has moved far enough to cause the slider level, as expressed in application terms, to change. If a user were to slowly drag the knob of this color slider all the way to the right, the program will only hear messages for levels 12, 13, 14 and 15, with an optional additional message when the user releases the mouse-button.

Changing the current level of the slider from within the program is as simple as specifying the new level in a function call. For instance, the application might set the slider's value to 5.

As a final point, the slider is very well-behaved. When the user releases the mouse-button, the slider immediately snaps to the centered position for the level. If a user sets their background color to light gray, which might have red = green = blue = 10, all three color sliders will have their knobs at precisely the same relative position, instead of anywhere in the range that means "ten".

- The NewGadget Structure
- Creating Gadgets
- Handling Gadget Messages
- IDCMP Flags
- Freeing Gadgets
- Simple GadTools Gadget Example
- Modifying Gadgets
- The Kinds of GadTools Gadgets
- Functions for Setting Up GadTools Menus and Gadgets
- Creating Gadget Lists
- Gadget Refresh Functions
- Other GadTools Functions
- Gadget Keyboard Equivalents
- Complete GadTools Gadget Example
- Restrictions on GadTools Gadgets
- Documented Side-Effects

1.15 15 / GadTools Gadgets / The NewGadget Structure

For most gadgets, the NewGadget structure is used to specify its common attributes. Additional attributes that are unique to specific kinds of gadgets are specified as tags sent to the CreateGadget() function (described below).

The NewGadget structure is defined in <libraries/gadtools.h> as:

```
struct NewGadget
{
    WORD ng_LeftEdge, ng_TopEdge;
    WORD ng_Width, ng_Height;
    UBYTE *ng_GadgetText;
    struct TextAttr *ng_TextAttr;
    WORD ng_GadgetID;
    ULONG ng_Flags;
    APTR ng_VisualInfo;
    APTR ng_UserData;
```

```
};
```

The fields of the `NewGadget` structure are used as follows:

`ng_LeftEdge, ng_TopEdge`

Define the position of the gadget being created.

`ng_Width` and `ng_Height`

Define the size of the gadget being created.

`ng_GadgetText`

Most gadgets have an associated label, which might be the text in a button or beside a checkmark. This field contains a pointer to the appropriate string. Note that only the pointer to the text is copied, the text itself is not. The string supplied must remain constant and valid for the life of the gadget.

`ng_TextAttr`

The application must specify a font to use for the label and any other text that may be associated with the gadget.

`ng_Flags`

Used to describe general aspects of the gadget, which includes where the label is to be placed and whether the label should be rendered in the highlight color. The label may be positioned on the left side, the right side, centered above, centered below or dead-center on the gadget. For most gadget kinds, the label is placed on the left side by default, exceptions will be noted.

`ng_GadgetID, ng_UserData`

These user fields are copied into the resulting Gadget structure.

`ng_VisualInfo`

This field must contain a pointer to an instance of the `VisualInfo` structure, which contains information needed to create and render GadTools gadgets. The `VisualInfo` structure itself is private to GadTools and subject to change. Use the specialized GadTools functions for accessing the `VisualInfo` pointer, defined below. Never access or modify fields within this structure.

1.16 15 / GadTools Gadgets / Creating Gadgets

The main call used to create a gadget with GadTools is `CreateGadget()`. This function can be used to create a single gadget or it can be called repeatedly to create a linked list of gadgets. It takes three arguments followed by a set of tags:

```
struct Gadget *CreateGadget( ULONG kind, struct Gadget *prevgad,
                             struct NewGadget *newgad,
                             struct TagItem *taglist)
struct Gadget *CreateGadgetA(ULONG kind, struct Gadget *prevgad,
                             struct NewGadget *newgad,
                             struct Tag tag1, ...)
```

Set the `kind` argument to one of the 12 gadget types supported by GadTools.

Set the `prevgad` argument to the gadget address returned by `CreateContext()` if this is the first (or only) gadget in the list. Subsequent calls to `CreateGadget()` can be used to create and link AutoDocs/gadgets together in a list in which case the `prevgad` argument is set to the address of the gadget returned by the preceding call to `CreateGadget()`.

Set the `newgad` argument to the address of the `NewGadget` structure describing the gadget to be created and set any special attributes for this gadget type using the tag arguments, `tag1` or `taglist`. For instance, the following code fragment might be used to create the color slider discussed earlier:

```
slidergad = CreateGadget(SLIDER_KIND, newgad, prevgad,
    GTSL_Min, 0,
    GTSL_Max, 15,
    GTSL_Level, 11,
    TAG_END);
```

`CreateGadget()` typically allocates and initializes all the necessary Intuition structures, including in this case the `Gadget`, `IntuiText` and `PropInfo` structures, as well as certain buffers. For more about these underlying structures, see the "Intuition Gadgets" chapter.

Since `CreateGadget()` is a tag-based function, it is easy to add more tags to get a fancier gadget. For example, `GadTools` can optionally display the running level beside the slider. The caller must supply a `printf()`-style formatting string and the maximum length that the string will resolve to when the number is inserted:

```
slidergad = CreateGadget(SLIDER_KIND, newgad, prevgad,
    GTSL_Min, 0,
    GTSL_Max, 15,
    GTSL_Level, 11,
    GTSL_LevelFormat, "%2ld" /* printf()-style formatting string */
    GTSL_MaxLevelLen, 2,    /* maximum length of string          */
    TAG_END);
```

The level, 0 to 15 in this example, would then be displayed beside the slider. The formatting string could instead be `"%2ld/15"`, so the level would be displayed as `"0/15"` through `"15/15"`.

1.17 15 / GadTools Gadgets / Handling Gadget Messages

`GadTools` gadgets follow the same input model as other Intuition components. When the user operates a `GadTools` gadget, Intuition notifies the application about the input event by sending an `IntuiMessage`. The application can get these messages at the `Window.UserPort`. However `GadTools` gadgets use different message handling functions to get and reply these messages. Instead of the `Exec` functions `GetMsg()` and `ReplyMsg()`, applications should get and reply these messages through a pair of special `GadTools` functions, `GT_GetIMsg()` and `GT_ReplyIMsg()`.

```
struct IntuiMessage *GT_GetIMsg(struct MsgPort *iport)
void GT_ReplyIMsg(struct IntuiMessage *imsg)
```

For `GT_GetIMsg()`, the `iport` argument should be set to the window's `UserPort`. For `GT_ReplyIMsg()`, the `imsg` argument should be set to a pointer to the `IntuiMessage` returned by `GT_GetIMsg()`.

These functions ensure that the application only sees the gadget events that concern it and in a desirable form. For example, with a GadTools slider gadget, a message only gets through to the application when the slider's level actually changes and that level can be found in the `IntuiMessage`'s `Code` field:

```
imsg = GT_GetIMsg(win->UserPort);
object = imsg->IAddress;
class = imsg->Class;
code = imsg->Code;
GT_ReplyIMsg(imsg);
switch (class)
{
    case IDCMP_MOUSEMOVE:
        if (object == slidergad)
        {
            printf("Slider at level %ld\n", code);
        }
        ...
        break;
    ...
}
```

In general, the `IntuiMessages` received from GadTools contain more information in the `Code` field than is found in regular Intuition gadget messages. Also, when dealing with GadTools a lot of messages (mostly `IDCMP_MOUSEMOVEs`) do not have to be processed by the application. These are two reasons why dealing with GadTools gadgets is much easier than dealing with regular Intuition gadgets. Unfortunately this processing cannot happen magically, so applications must use `GT_GetIMsg()` and `GT_ReplyIMsg()` where they would normally have used `GetMsg()` and `ReplyMsg()`.

`GT_GetIMsg()` actually calls `GetMsg()` to remove a message from the specified window's `UserPort`. If the message pertains to a GadTools gadget then some dispatching code in GadTools will be called to process the message. What the program will receive from `GT_GetIMsg()` is actually a copy of the real `IntuiMessage`, possibly with some supplementary information from GadTools, such as the information typically found in the `Code` field.

The `GT_ReplyIMsg()` call will take care of cleaning up and replying to the real `IntuiMessage`.

Warning:

When an `IDCMP_MOUSEMOVE` message is received from a GadTools gadget, GadTools arranges to have the gadget's pointer in the `IAddress` field of the `IntuiMessage`. While this is extremely convenient, it is also untrue of messages from regular Intuition gadgets (described in the "Intuition Gadgets" chapter). Do not make the mistake of assuming it to be true.

This description of the inner workings of `GT_GetIMsg()` and `GT_ReplyIMsg()`

is provided for understanding only; it is crucial that the program make no assumptions or interpretations about the real `IntuiMessage`. Any such inferences are not likely to hold true in the future. See the section on documented side-effects for more information.

1.18 15 / GadTools Gadgets / IDCMP Flags

The various GadTools gadget types require certain classes of IDCMP messages in order to work. Applications specify these IDCMP classes when the window is opened or later with `ModifyIDCMP()` (see "Intuition Windows" chapter for more on this). Each kind of GadTools gadget requires one or more of these IDCMP classes: `IDCMP_GADGETUP`, `IDCMP_GADGETDOWN`, `IDCMP_MOUSEMOVE`, `IDCMP_MOUSEBUTTONS` and `IDCMP_INTUITICKS`. As a convenience, the IDCMP classes required by each kind of gadget are defined in `<libraries/gadtools.h>`. For example, `SLIDERIDCMP` is defined to be:

```
#define SLIDERIDCMP (IDCMP_GADGETUP | IDCMP_GADGETDOWN |
                    IDCMP_MOUSEMOVE)
```

Always OR the IDCMP Flag Bits.

When specifying the IDCMP classes for a window, never add the flags together, always OR the bits together. Since many of the GadTools IDCMP constants have multiple bits set, adding the values will not lead to the proper flag combination.

If a certain kind of GadTools gadget is used, the window must use all IDCMP classes required by that kind of gadget. Do not omit any that are given for that class, even if the application does require the message type.

Because of the way GadTools gadgets are implemented, programs that use them always require notification about window refresh events. Even if the application performs no rendering of its own, it may not use the `WFLG_NOCAREREFRESH` window flag and must always set `IDCMP_REFRESHWINDOW`. See the section on "Gadget Refresh Functions" later in this chapter for more on this.

1.19 15 / GadTools Gadgets / Freeing Gadgets

After closing the window, the gadgets allocated using `CreateGadget()` must be released. `FreeGadgets()` is a simple call that will free all the GadTools gadgets that it finds, beginning with the gadget whose pointer is passed as an argument.

```
void FreeGadgets( struct Gadget *gad );
```

The `gad` argument is a pointer to the first gadget to be freed. It is safe to call `FreeGadgets()` with a `NULL` gadget pointer, the function will then return immediately. Before calling `FreeGadgets()`, the application must first either remove the gadgets or close the window.

When the gadget passed to `FreeGadgets()` is the first gadget in a linked list, the function frees all the GadTools gadgets on the list without patching pointers or trying to maintain the integrity of the list. Any non-GadTools gadgets found on the list will not be freed, hence the result will not necessarily form a nice list since any intervening GadTools gadgets will be gone.

See the section on "Creating Gadget Lists" for more information on using linked lists of gadgets.

1.20 15 / GadTools Gadgets / Modifying Gadgets

The attributes of a gadget are set up when the gadget is created. Some of these attributes can be changed later by using the `GT_SetGadgetAttrs()` function:

```
void GT_SetGadgetAttrs (struct Gadget *gad, struct Window *win,
                        struct Requester *req, Tag tag1, ... )
void GT_SetGadgetAttrsA(struct Gadget *gad, struct Window *win,
                        struct Requester *req, struct TagItem *taglist)
```

The `gad` argument specifies the gadget to be changed while the `win` argument specifies the window the gadget is in. Currently, the `req` argument is unused and must be set to `NULL`.

The gadget attributes are changed by passing tag arguments to these functions. The tag arguments can be either a set of `TagItems` on the stack for `GT_SetGadgetAttrs()`, or a pointer to an array of `TagItems` for `GT_SetGadgetAttrsA()`. The tag items specify the attributes that are to be changed for the gadget. Keep in mind though that not every gadget attribute can be modified this way.

For example, in the slider gadget presented earlier, the level-formatting string may not be changed after the gadget is created. However, the slider's level may be changed to 5 as follows:

```
GT_SetGadgetAttrs(slidergad, win, req,
    GTSL_Level, 5,
    TAG_END);
```

Here are some other example uses of `GT_SetGadgetAttrs()` to change gadget attributes after it is created.

```
/* Disable a button gadget */
GT_SetGadgetAttrs(buttongad, win, NULL,
    GA_Disabled, TRUE,
    TAG_END);

/* Change a slider's range to be 1 to 100, currently at 50 */
GT_SetGadgetAttrs(slidergad, win, NULL,
    GTSL_Min, 1,
    GTSL_Max, 100,
    GTSL_Level, 50,
    TAG_END);
```

```

/* Add a node to the head of listview's list, and make it */
/* the selected one */
GT_SetGadgetAttrs(listviewgad, win, NULL,
                  /* detach list before modifying */
                  GTLV_Labels, ~0,
                  TAG_END);
AddHead(&lvlabels, &newnode);
GT_SetGadgetAttrs(listviewgad, win, NULL,
                  /* re-attach list */
                  GTLV_Labels, &lvlabels,
                  GTLV_Selected, 0,
                  TAG_END);

```

When changing a gadget using these functions, the gadget will automatically update its visuals. No refresh is required, nor should any refresh call be performed.

Warning:

The `GT_SetGadgetAttrs()` functions may not be called inside of a `GT_BeginRefresh()/GT_EndRefresh()` pair. This is true of Intuition gadget functions generally, including those discussed in the "Intuition Gadgets" chapter.

In the sections that follow all the possible attributes for each kind of gadget are discussed. The tags are also described in the Autodocs for `GT_SetGadgetAttrs()` in the Amiga ROM Kernel Reference Manual: Includes and Autodocs.

Important:

Tags that can only be sent to `CreateGadget()` and not to `GT_SetGadgetAttrs()` will be marked as create only in the discussion that follows. Those that are valid parameters to both functions will be marked as create and set.

1.21 15 / GadTools Gadgets / The Kinds of GadTools Gadgets

This section discusses the unique features of each kind of gadget supported by the GadTools library.

- Button Gadgets
- Text-Entry and Number-Entry Gadgets
- Checkbox Gadgets
- Mutually-Exclusive Gadgets
- Cycle Gadgets
- Slider Gadgets
- Scroller Gadgets
- Listview Gadgets
- Palette Gadgets
- Text-Display and Numeric-Display Gadgets
- Generic Gadgets

1.22 15 // The Kinds of GadTools Gadgets / Button Gadgets

Button gadgets (BUTTON_KIND) are perhaps the simplest kind of GadTools gadget. Button gadgets may be used for objects like the "OK" and "Cancel" buttons in requesters. GadTools will create a hit-select button with a raised bevelled border. The label supplied will be centered on the button's face. Since the label is not clipped, be sure that the gadget is large enough to contain the text supplied.

Button gadgets recognize only one tag:

GA_Disabled (BOOL)

Set this attribute to TRUE to disable or ghost the button gadget, to FALSE otherwise. The default is FALSE. (Create and set.)

When the user selects a button gadget, the program will receive an IDCMP_GADGETUP event.

If clicking on a button causes a requester to appear, for example a button that brings up a color requester, then the button text should end in ellipsis (...), as in "Quit..."

1.23 15 // Kinds of GadTools Gadgets / Text-Entry and Number-Entry Gadgets

Text-entry (STRING_KIND) and number-entry (INTEGER_KIND) gadgets are fairly typical Intuition string gadgets. The typing area is contained by a border which is a raised ridge.

Text-entry gadgets accept the following tags:

GTST_String (STRPTR)

A pointer to the string to be placed into the text-entry gadget buffer or NULL to get an empty text-entry gadget. The string itself is actually copied into the gadget's buffer. The default is NULL. (Create and set.)

GTST_MaxChars (UWORD)

The maximum number of characters that the text-entry gadget should hold. The string buffer that gets created for the gadget will actually be one bigger than this number, in order to hold the trailing NULL. The default is 64. (Create only.)

Number-entry gadgets accept the following tags:

GTIN_Number (ULONG)

The number to be placed into the number-entry gadget. The default is zero. (Create and set.)

GTIN_MaxChars (UWORD)

The maximum number of digits that the number-entry gadget should hold. The string buffer that gets created for the gadget will actually be one bigger than this, in order to hold the trailing NULL. The default is 10. (Create only.)

Both text-entry and number-entry gadgets, which are collectively called string gadgets, accept these common tags:

STRINGA_Justification

This attribute controls the placement of the string or number within its box and can be one of GACT_STRINGLEFT, GACT_STRINGRIGHT or GACT_STRINGCENTER. The default is GACT_STRINGLEFT. (Create only.)

STRINGA_ReplaceMode (BOOL)

Set STRINGA_ReplaceMode to TRUE to get a string gadget which is in replace-mode, as opposed to auto-insert mode. (Create only.)

GA_Disabled (BOOL)

Set this attribute to TRUE to disable the string gadget, otherwise to FALSE. The default is FALSE. (Create and set.)

STRINGA_ExitHelp (BOOL)

(New for V37, ignored under V36). Set this attribute to TRUE if the application wants to hear the Help key from within this string gadget. This feature allows the program to hear the press of the Help key in all cases. If TRUE, pressing the help key while this gadget is active will terminate the gadget and send a message. The program will receive an IDCMP_GADGETUP message having a Code value of 0x5F, the rawkey code for Help. Typically, the program will want to reactivate the gadget after performing the help-display. The default is FALSE. (Create only.)

GA_TabCycle (BOOL)

(New for V37, ignored under V36). If the user types Tab or Shift Tab into a GA_TabCycle gadget, Intuition will activate the next or previous such gadget in sequence. This gives the user easy keyboard control over which text-entry or number-entry gadget is active. Tab moves to the next GA_TabCycle gadget in the gadget list and Shift Tab moves to the previous one. When the user presses Tab or Shift Tab, Intuition will deactivate the gadget and send this program an IDCMP_GADGETUP message with the code field set to 0x09, the ASCII value for a tab. Intuition will then activate the next indicated gadget. Check the shift bits of the qualifier field to learn if Shift Tab was typed. The ordering of the gadgets may only be controlled by the order in which they were added to the window. For special cases, for example, if there is only one string gadget in the window, this feature can be suppressed by specifying the tagitem pair {GA_TabCycle, FALSE}. The default is TRUE. (Create only.)

GTST_EditHook (struct Hook *)

(New for V37, ignored under V36). Pointer to a custom editing hook for this string or integer gadget. See the "Intuition Gadgets" chapter for more information on string gadget edit-hooks.

As with all Intuition string gadgets, the program will receive an IDCMP_GADGETUP message only when the user presses Enter, Return, Help, Tab or Shift Tab while typing in the gadget. Note that, like Intuition string gadgets, the program will not hear anything if the user deactivates the string gadget by clicking elsewhere. Therefore, it is a good idea to always check the string gadget's buffer before using its contents, instead

of just tracking its value as IDCMP_GADGETUP messages are received for this gadget.

Be sure the code is designed so that nothing drastic happens, like closing a requester or opening a file, if the IDCMP_GADGETUP message has a non-zero Code field; the program will want to handle the Tab and Help cases intelligently.

To read the string gadget's buffer, look at the Gadget's StringInfo Buffer:

```
((struct StringInfo *)gad->SpecialInfo)->Buffer
```

To determine the value of an integer gadget, look at the Gadget's StringInfo LongInt in the same way.

Always use the GTST_String or GTIN_Number tags to set these values. Never write to the StringInfo->Buffer or StringInfo->LongInt fields directly.

GadTools string and integer gadgets do not directly support the GA_Immediate property (which would cause Intuition to send an IDCMP_GADGETDOWN event when such a gadget is first selected). However, this property can be very important. Therefore, the following technique can be used to enable this property.

Warning:

Note that the technique shown here relies on directly setting flags in a GadTools gadget; this is not normally allowed since it hinders future compatibility. Do not attempt to change other flags or properties of GadTools gadgets except through the defined interfaces of CreateGadgetA() and GT_SetGadgetAttrs(). Directly modifying flags or properties is legal only when officially sanctioned by Amiga, Inc..

To get the GA_Immediate property, pass the {GA_Immediate,TRUE} tag to CreateGadgetA(). Even though this tag is ignored for string and integer gadgets under V37, this will allow future versions of GadTools to learn of your request in the correct way. Then, under V37 only, set the GACT_IMMEDIATE flag in the gadget's Activation field:

```
gad = CreateGadget( STRING_KIND, gad, &ng,
    /* string gadget tags go here */
    GTST_...,

    /* Add this tag for future GadTools releases */
    GA_Immediate, TRUE,
    ...
    TAG_DONE );

if ( ( gad ) && ( GadToolsBase->lib_Version == 37) )
{
    /* Under GadTools V37 only, set this attribute
     * directly. Do not set this attribute under
     * future versions of GadTools, or for gadgets
     * other than STRING_KIND or INTEGER_KIND.
     */
    gad->Activation |= GACT_IMMEDIATE;
```

```
}
```

1.24 15 // The Kinds of GadTools Gadgets / Checkbox Gadgets

Checkboxes (CHECKBOX_KIND) are appropriate for presenting options which may be turned on or off. This kind of gadget consists of a raised box which contains a checkmark if the option is selected or is blank if the option is not selected. Clicking on the box toggles the state of the checkbox.

The width and height of a checkbox are currently fixed (to 26x11). If variable-sized checkboxes are added in the future, they will be done in a compatible manner. Currently the width and height specified in the NewGadget structure are ignored.

The checkbox may be controlled with the following tags:

GTCB_Checked (BOOL)

Set this attribute to TRUE to set the gadget's state to checked. Set it to FALSE to mark the gadget as unchecked. The default is FALSE. (Create and set.)

GA_Disabled (BOOL)

Set this attribute to TRUE to disable the checkbox, to FALSE otherwise. The default is FALSE. (Create and set.)

When the user selects a checkbox, the program will receive an IntuiMessage with a class of IDCMP_GADGETUP. As this gadget always toggles, the program can easily track the state of the gadget. Feel free to read the Gadget->Flags GFLG_SELECTED bit. Note, however, that the Gadget structure itself is not synchronized to the IntuiMessages received. If the user clicks a second time, the GFLG_SELECTED bit can toggle again before the program gets a chance to read it. This is true of any of the dynamic fields of the Gadget structure, and is worth being aware of, although only rarely will an application have to account for it.

1.25 15 // The Kinds of GadTools Gadgets / Mutually-Exclusive Gadgets

Use mutually exclusive gadgets (MX_KIND), or radio buttons, when the user must choose only one option from a short list of possibilities. Mutually exclusive gadgets are appropriate when there are a small number of choices, perhaps eight or less.

A set of mutually exclusive gadgets consists of a list of labels and beside each label, a small raised oval that looks like a button. Exactly one of the ovals is recessed and highlighted, to indicate the selected choice. The user can pick another choice by clicking on any of the raised ovals. This choice will become active and the previously selected choice will become inactive. That is, the selected oval will become recessed while the previous one will pop out, like the buttons on a car radio.

Mutually exclusive gadgets recognize these tags:

GTMX_Labels (STRPTR *)

A NULL-pointer-terminated array of strings which are to be the labels beside each choice in the set of mutually exclusive gadgets. This array determines how many buttons are created. This array must be supplied to `CreateGadget()` and may not be changed. The strings themselves must remain valid for the lifetime of the gadget. (Create only.)

GTMX_Active (UWORD)

The ordinal number, counting from zero, of the active choice of the set of mutually exclusive gadgets. The default is zero. (Create and set.)

GTMX_Spacing (UWORD)

The amount of space, in pixels, that will be placed between successive choices in a set of mutually exclusive gadgets. The default is one. (Create only.)

When the user selects a new choice from a set of mutually exclusive gadgets, the program will receive an `IDCMP_GADGETDOWN` `IntuiMessage`. Look in the `IntuiMessage`'s `Code` field for the ordinal number of the new active selection.

The `ng_GadgetText` field of the `NewGadget` structure is ignored for mutually exclusive gadgets. The text position specified in `ng_Flags` determines whether the item labels are placed to the left or the right of the radio buttons themselves. By default, the labels appear on the left. Do not specify `PLACETEXT_ABOVE`, `PLACETEXT_BELOW` or `PLACETEXT_IN` for this kind of gadget.

Like the checkbox, the size of the radio button is currently fixed, and the dimensions supplied in the `NewGadget` structure are ignored. If in the future the buttons are made scalable, it will be done in a compatible manner. Currently, mutually exclusive gadgets may not be disabled.

1.26 15 // The Kinds of GadTools Gadgets / Cycle Gadgets

Like mutually exclusive gadgets, cycle gadgets (`CYCLE_KIND`) allow the user to choose exactly one option from among several.

The cycle gadget appears as a raised rectangular button with a vertical divider near the left side. A circular arrow glyph appears to the left of the divider, while the current choice appears to the right. Clicking on the cycle gadget advances to the next choice, while shift-clicking on it changes it to the previous choice.

Cycle gadgets are more compact than mutually exclusive gadgets, since only the current choice is displayed. They are preferable to mutually exclusive gadgets when a window needs to have several such gadgets as in the `PrinterGfx` Preferences editor, or when there is a medium number of choices. If the number of choices is much more than about a dozen, it may become too frustrating and inefficient for the user to find the desired choice. In that case, use a listview (scrolling list) instead.

The tags recognized by cycle gadgets are:

GTCY_Labels (STRPTR *)

Like GTMX_Labels, this tag is associated with a NULL-pointer-terminated array of strings which are the choices that this gadget allows. This array must be supplied to CreateGadget(), and can only be changed starting in V37. The strings themselves must remain valid for the lifetime of the gadget. (Create only (V36), Create and set (V37).)

GTCY_Active (UWORD)

The ordinal number, counting from zero, of the active choice of the cycle gadget. The default is zero. (Create and set.)

GA_Disabled (BOOL)

(New for V37, ignored by V36.) Set this attribute to TRUE to disable the cycle gadget, to FALSE otherwise. The default is FALSE. (Create and set.)

When the user clicks or shift-clicks on a cycle gadget, the program will receive an IDCMP_GADGETUP IntuiMessage. Look in the Code field of the IntuiMessage for the ordinal number of the new active selection.

1.27 15 // The Kinds of GadTools Gadgets / Slider Gadgets

Sliders are one of the two kinds of proportional gadgets offered by GadTools. Slider gadgets (SLIDER_KIND) are used to control an amount, a level or an intensity, such as volume or color. Scroller gadgets (SCROLLER_KIND) are discussed below.

Slider gadgets accept the following tags:

GTSL_Min (WORD)

The minimum level of a slider. The default is zero. (Create and set.)

GTSL_Max (WORD)

The maximum level of a slider. The default is 15. (Create and set.)

GTSL_Level (WORD)

The current level of a slider. The default is zero. When the level is at its minimum, the knob will be all the way to the left for a horizontal slider or all the way at the bottom for a vertical slider. Conversely, the maximum level corresponds to the knob being to the extreme right or top. (Create and set.)

GTSL_LevelFormat (STRPTR)

The current level of the slider may be displayed in real-time alongside the gadget. To use the level-display feature, the program must be using a monospace font for this gadget.

GTSL_LevelFormat specifies a printf()-style formatting string used to render the slider level beside the slider (the complete set of formatting options is described in the Exec library function RawDoFmt()). Be sure to use the 'l' (long word) modifier for the

number. Field-width specifiers may be used to ensure that the resulting string is always of constant width. The simplest would be "%2ld". A 2-digit hexadecimal slider might use "%02lx", which adds leading zeros to the number. Strings with extra text, such as "%3ld hours", are permissible. If this tag is specified, the program must also provide GTSL_MaxLevelLen. By default, the level is not displayed. (Create only.)

GTSL_MaxLevelLen (UWORD)

The maximum length of the string that will result from the given level-formatting string. If this tag is specified, the program must also provide GTSL_LevelFormat. By default, the level is not displayed. (Create only.)

GTSL_LevelPlace

To choose where the optional display of the level is positioned. It must be one of PLACETEXT_LEFT, PLACETEXT_RIGHT, PLACETEXT_ABOVE or PLACETEXT_BELOW. The level may be placed anywhere with the following exception: the level and the label may not be both above or both below the gadget. To place them both on the same side, allow space in the gadget's label (see the example). The default is PLACETEXT_LEFT. (Create only.)

GTSL_DispFunc (LONG (*function)(struct Gadget *, WORD))

Optional function to convert the level for display. A slider to select the number of colors for a screen may operate in screen depth (1 to 5, for instance), but actually display the number of colors (2, 4, 8, 16 or 32). This may be done by providing a GTSL_DispFunc function which returns $1 \ll \text{level}$. The function must take a pointer to the Gadget as the first parameter and the level, a WORD, as the second and return the result as a LONG. The default behavior for displaying a level is to do so without any conversion. (Create only.)

GA_Immediate (BOOL)

Set this to TRUE to receive an IDCMP_GADGETDOWN IntuiMessage when the user presses the mouse button over the slider. The default is FALSE. (Create only.)

GA_RelVerify (BOOL)

Set this to TRUE to receive an IDCMP_GADGETUP IntuiMessage when the user releases the mouse button after using the slider. The default is FALSE. (Create only.)

PGA_Freedom

Specifies which direction the knob may move. Set to LORIENT_VERT for a vertical slider or LORIENT_HORIZ for a horizontal slider. The default is LORIENT_HORIZ. (Create only.)

GA_Disabled (BOOL)

Set this attribute to TRUE to disable the slider, to FALSE otherwise. The default is FALSE. (Create and set.)

Up to three different classes of IntuiMessage may be received at the port when the user plays with a slider, these are IDCMP_MOUSEMOVE, IDCMP_GADGETUP and IDCMP_GADGETDOWN. The program may examine the IntuiMessage Code field to discover the slider's level.

IDCMP_MOUSEMOVE IntuiMessages will be heard whenever the slider's level changes. IDCMP_MOUSEMOVE IntuiMessages will not be heard if the knob has not moved far enough for the level to actually change. For example if the slider runs from 0 to 15 and is currently set to 12, if the user drags the slider all the way up the program will hear no more than three IDCMP_MOUSEMOVES, one each for 13, 14 and 15.

If {GA_Immediate, TRUE} is specified, then the program will always hear an IDCMP_GADGETDOWN IntuiMessage when the user begins to adjust a slider. If {GA_RelVerify, TRUE} is specified, then the program will always hear an IDCMP_GADGETUP IntuiMessage when the user finishes adjusting the slider. If IDCMP_GADGETUP or IDCMP_GADGETDOWN IntuiMessages are requested, the program will always hear them, even if the level has not changed since the previous IntuiMessage.

Note that the Code field of the IntuiMessage structure is a UWORD, while the slider's level may be negative, since it is a WORD. Be sure to copy or cast the IntuiMessage->Code into a WORD if the slider has negative levels.

If the user clicks in the container next to the knob, the slider level will increase or decrease by one. If the user drags the knob itself, then the knob will snap to the nearest integral position when it is released.

Here is an example of the screen-depth slider discussed earlier:

```
/* NewGadget initialized here. Note the three spaces
 * after "Slider:", to allow a blank plus the two digits
 * of the level display
 */
ng.ng_Flags = PLACETEXT_LEFT;
ng.ng_GadgetText = "Slider:  ";

LONG DepthToColors(struct Gadget *gad, WORD level)
{
    return ((WORD)(1 << level));
}

...

gad = CreateGadget(SLIDER_KIND, gad, &ng,
    GTSL_Min, 1,
    GTSL_Max, 5,
    GTSL_Level, current_depth,
    GTSL_MaxLevelLen, 2,
    GTSL_LevelFormat, "%2ld",
    GTSL_DispFunc, DepthToColors,
    TAG_END);
```

1.28 15 // The Kinds of GadTools Gadgets / Scroller Gadgets

Scrollers (SCROLLER_KIND) bear some similarity to sliders, but are used for a quite different job: they allow the user to adjust the position of a limited view into a larger area. For example, Workbench's windows have scrollers that allow the user to see icons that are outside the visible

portion of a window. Another example is a scrolling list in a file requester which has a scroller that allows the user to see different parts of the whole list.

A scroller consists of a proportional gadget and usually also has a pair of arrow buttons.

While the slider deals in minimum, maximum and current level, the scroller understands Total, Visible and Top. For a scrolling list, Total would be the number of items in the entire list, Visible would be the number of lines visible in the display area and Top would be the number of the first line displayed in the visible part of the list. Top would run from zero to Total - Visible. For an area-scroller such as those in Workbench's windows, Total would be the height (or width) of the whole area, Visible would be the visible height (or width) and Top would be the top (or left) edge of the visible part.

Note that the position of a scroller should always represent the position of the visible part of the project and never the position of a cursor or insertion point.

Scrollers respect the following tags:

GTSC_Top (WORD)

The top line or position visible in the area that the scroller represents. The default is zero. (Create and set.)

GTSC_Total (WORD)

The total number of lines or positions that the scroller represents. The default is zero. (Create and set.)

GTSC_Visible (WORD)

The visible number of lines or positions that the scroller represents. The default is two. (Create and set.)

GTSC_Arrows (UWORD)

Asks for arrow gadgets to be attached to the scroller. The value supplied will be used as the width of each arrow button for a horizontal scroller or the height of each arrow button for a vertical scroller, the other dimension will be set by GadTools to match the scroller size. It is generally recommend that arrows be provided. The default is no arrows. (Create only.)

GA_Immediate (BOOL)

Set this to TRUE to receive an IDCMP_GADGETDOWN IntuiMessage when the user presses the mouse button over the scroller. The default is FALSE. (Create only.)

GA_RelVerify (BOOL)

Set this to TRUE to receive an IDCMP_GADGETUP IntuiMessage when the user releases the mouse button after using the scroller. The default is FALSE. (Create only.)

PGA_Freedom

Specifies which direction the knob may move. Set to LORIENT_VERT for a vertical scroller or LORIENT_HORIZ for a horizontal scroller. The default is LORIENT_HORIZ. (Create only.)

GA_Disabled (BOOL)

Set this attribute to TRUE to disable the scroller, to FALSE otherwise. The default is FALSE. (Create and set.)

The IntuiMessages received for a scroller gadget are the same in nature as those for a slider defined above, including the fact that messages are only heard by the program when the knob moves far enough for the Top value to actually change. The Code field of the IntuiMessage will contain the new Top value of the scroller.

If the user clicks on an arrow gadget, the scroller moves by one unit. If the user holds the button down over an arrow gadget, it repeats.

If the user clicks in the container next to the knob, the scroller will move by one page, which is the visible amount less one. This means that when the user pages through a scrolling list, any pair of successive views will overlap by one line. This helps the user understand the continuity of the list. If the program is using a scroller to pan through an area then there will be an overlap of one unit between successive views. It is recommended that Top, Visible and Total be scaled so that one unit represents about five to ten percent of the visible amount.

1.29 15 // The Kinds of GadTools Gadgets / Listview Gadgets

Listview gadgets (LISTVIEW_KIND) are scrolling lists. They consist of a scroller with arrows, an area where the list itself is visible and optionally a place where the current selection is displayed, which may be editable. The user can browse through the list using the scroller or its arrows and may select an entry by clicking on that item.

There are a number of tags that are used with listviews:

GTLV_Labels (struct List *)

An Exec list whose nodes' ln_Name fields are to be displayed as items in the scrolling list. If the list is empty, an empty List structure or a NULL value may be used for GTLV_Labels. This tag accepts a value of "~0" to detach the list from the listview, defined below. The default is NULL. (Create and set.)

GTLV_Top (UWORD)

The ordinal number of the top item visible in the listview. The default is zero. (Create and set.)

GTLV_ReadOnly (BOOL)

Set this to TRUE for a read-only listview, which the user can browse, but not select items from. A read-only listview can be recognized because the list area is recessed, not raised. The default is FALSE. (Create only.)

GTLV_ScrollWidth (UWORD)

The width of the scroller to be used in the listview. Any value specified must be reasonably bigger than zero. The default is 16. (Create only.)

GTLV_ShowSelected (struct Gadget *)

Use this tag to show the currently selected entry displayed underneath the listview. Set its value to NULL to get a read-only (TEXT_KIND) display of the currently selected entry or set it to a pointer to an already-created GadTools STRING_KIND gadget to allow the user to directly edit the current entry. By default, there is no display of the currently selected entry. (Create only.)

GTLV_Selected (UWORD)

Ordinal number of the item to be placed into the display of the current selection under the listview. This tag is ignored if GTLV_ShowSelected is not used. Set it to "~0" to have no current selection. The default is "~0". (Create and set.)

LAYOUTA_Spacing (UWORD)

Extra space, in pixels, to be placed between the entries in the listview. The default is zero. (Create only.)

The program will only hear from a listview when the user selects an item from the list. The program will then receive an IDCMP_GADGETUP IntuiMessage. This message will contain the ordinal number of the item within the list that was selected in the Code field of the message. This number is independent of the displayed listview, it is the offset from the start of the list of items.

If the program attaches a display gadget by using the TagItem {GTLV_ShowSelected, NULL}, then whenever the user clicks on an entry in the listview it will be copied into the display gadget.

If the display gadget is to be editable, then the program must first create a GadTools STRING_KIND gadget whose width matches the width of the listview. The TagItem {GTLV_ShowSelected, stringgad} is used to install the editable gadget, where stringgad is the pointer returned by CreateGadget(). When the user selects any entry from the listview, it gets copied into the string gadget. The user can edit the string and the program will hear normal string gadget IDCMP_GADGETUP messages from the STRING_KIND gadget.

The Exec List and its Node structures may not be modified while they are attached to the listview, since the list might be needed at any time. If the program has prepared an entire new list, including a new List structure and all new nodes, it may replace the currently displayed list in a single step by calling GT_SetGadgetAttrs() with the TagItem {GTLV_Labels, newlist}. If the program needs to operate on the list that has already been passed to the listview, it should detach the list by setting the GTLV_Labels attribute to "~0". When done modifying the list, resubmit it by setting GTLV_Labels to once again point to it. This is better than first setting the labels to NULL and later back to the list, since setting GTLV_Labels to NULL will visually clear the listview. If the GTLV_Labels attribute is set to "~0", the program is expected to set it back to something determinate, either a list or NULL, soon after.

The height specified for the listview will determine the number of lines in the list area. When creating a listview, it will be no bigger than the size specified in the NewGadget structure. The size will include the current-display gadget, if any, that has been requested via the GTLV_ShowSelected tag. The listview may end up being less tall than the

application asked for, since the calculated height assumes an integral number of lines in the list area.

By default, the gadget label will be placed above the listview. This may be overridden using `ng_Flags`.

Currently, a listview may not be disabled.

1.30 15 // The Kinds of GadTools Gadgets / Palette Gadgets

Palette gadgets (`PALETTE_KIND`) let the user pick a color from a set of several. A palette gadget consists of a number of colored squares, one for each color available. There may also be an optional indicator square which is filled with the currently selected color. To create a color editor, a palette gadget would be combined with some sliders to control red, green and blue components, for example.

Palette gadgets use the following tags:

`GTPA_Depth` (UWORD)

The number of bitplanes that the palette represents. There will be 1 << depth squares in the palette gadget. The default is one. (Create only.)

`GTPA_Color` (UBYTE)

The selected color of the palette. The default is one. (Create and set.)

`GTPA_ColorOffset` (UBYTE)

The first color to use in the palette. For example, if `GTPA_Depth` is two and `GTPA_ColorOffset` is four, then the palette will have squares for colors four, five, six and seven. The default is zero. (Create only.)

`GTPA_IndicatorWidth` (UWORD)

The desired width of the current-color indicator. By specifying this tag, the application is asking for an indicator to be placed to the left of the color selection squares. The indicator will be as tall as the gadget itself. By default there is no indicator. (Create only.)

`GTPA_IndicatorHeight` (UWORD)

The desired height of the current-color indicator. By specifying this tag, the application is asking for an indicator to be placed above the color selection squares. The indicator will be as wide as the gadget itself. By default there is no indicator. (Create only.)

`GA_Disabled` (BOOL)

Set this attribute to `TRUE` to disable the palette gadget, to `FALSE` otherwise. The default is `FALSE`. (Create and set.)

An `IDCMP_GADGETUP` `IntuiMessage` will be received when the user selects a color from the palette. The current-color indicator is recessed, indicating that clicking on it has no effect.

If the palette is wide and not tall, use the `GTPA_IndicatorWidth` tag to put the indicator on the left. If the palette is tall and narrow, put the indicator on top using `GTPA_IndicatorHeight`.

By default, the gadget's label will go above the palette gadget, unless `GTPA_IndicatorWidth` is specified, in which case the label will go on the left. In either case, the default may be overridden by setting the appropriate flag in the `NewGadget`'s `ng_Flags` field.

The size specified for the palette gadget will determine how the area is subdivided to make the individual color squares. The actual size of the palette gadget will be no bigger than the size given, but it can be smaller in order to make the color squares all exactly the same size.

1.31 15 // Kinds of GadTools / Text-Display and Numeric-Display Gadgets

Text-display (`TEXT_KIND`) and numeric-display (`NUMBER_KIND`) gadgets are read-only displays of information. They are useful for displaying information that is not editable or selectable, while allowing the application to use the GadTools formatting and visuals. Conveniently, the visuals are automatically refreshed through normal GadTools gadget processing. The values displayed may be modified by the program in the same way other GadTools gadgets may be updated.

Text-display and number-display gadgets consist of a fixed label (the one supplied as the `NewGadget`'s `ng_GadgetText`), as well as a changeable string or number (`GTTX_Text` or `GTNM_Number` respectively). The fixed label is placed according to the `PLACETEXT_` flag chosen in the `NewGadget` `ng_Flags` field. The variable part is aligned to the left-edge of the gadget.

Text-display gadgets recognize the following tags:

`GTTX_Text` (`STRPTR`)

Pointer to the string to be displayed or `NULL` for no string. The default is `NULL`. (Create and set.)

`GTTX_Border` (`BOOL`)

Set to `TRUE` to place a recessed border around the displayed string. The default is `FALSE`. (Create only.)

`GTTX_CopyText` (`BOOL`)

This flag instructs the text-display gadget to copy the supplied `GTTX_Text` string instead of using only a pointer to the string. This only works for the value of `GTTX_Text` set at `CreateGadget()` time. If `GTTX_Text` is changed, the new text will be referenced by pointer, not copied. Do not use this tag without a non-`NULL` `GTTX_Text`. (Create only.)

Number-display gadgets have the following tags:

`GTNM_Number` (`LONG`)

The number to be displayed. The default is zero. (Create or set.)

`GTNM_Border` (`BOOL`)

Set to TRUE to place a recessed border around the displayed number.
The default is FALSE. (Create only.)

Since they are not selectable, text-display and numeric-display gadgets never cause IntuiMessages to be sent to the application.

1.32 15 // The Kinds of GadTools Gadgets / Generic Gadgets

If the application requires a specialized gadget which does not fit into any of the defined GadTools kinds but would still like to use the GadTools gadget creation and deletion functions, it may create a GadTools generic gadget and use it any way it sees fit. In fact, all of the kinds of GadTools gadgets are created out of GadTools GENERIC_KIND gadgets.

The gadget that gets created will heed almost all the information contained in the NewGadget structure supplied.

If ng_GadgetText is supplied, the Gadget's GadgetText will point to an IntuiText structure with the provided string and font. However, do not specify any of the PLACETEXT ng_Flags, as they are currently ignored by GENERIC_KIND gadgets. PLACETEXT flags may be supported by generic GadTools gadgets in the future.

It is up to the program to set the Flags, Activation, GadgetRender, SelectRender, MutualExclude and SpecialInfo fields of the Gadget structure.

The application must also set the GadgetType field, but be certain to preserve the bits set by CreateGadget(). For instance, to make a gadget boolean, use:

```
gad->GadgetType |= GTYP_BOOLGADGET;
```

and not

```
gad->GadgetType = GTYP_BOOLGADGET;
```

Using direct assignment, (the = operator), clears all other flags in the GadgetType field and the gadget may not be properly freed by FreeGadgets().

1.33 15 / GadTools / Functions for Setting Up GadTools Menus and Gadgets

This section gives all the details on the functions used to set up GadTools menus and gadgets that were mentioned briefly earlier in this chapter.

```
GetVisualInfo() and FreeVisualInfo()      CreateContext()
```

1.34 15 /// GetVisualInfo() and FreeVisualInfo()

In order to ensure their best appearance, GadTools gadgets and menus need information about the screen on which they will appear. Before creating any GadTools gadgets or menus, the program must get this information using the `GetVisualInfo()` call.

```
APTR GetVisualInfoA( struct Screen *screen, struct TagItem *taglist );
APTR GetVisualInfo( struct Screen *screen, Tag tag1, ... );
```

Set the screen argument to a pointer to the screen you are using. The tag arguments, `tag1` or `taglist`, are reserved for future extensions. Currently none are recognized, so only `TAG_END` should be used.

The function returns an abstract handle called the `VisualInfo`. For GadTools gadgets, the `ng_VisualInfo` field of the `NewGadget` structure must be set to this handle before the gadget can be added to the window. GadTools menu layout and creation functions also require the `VisualInfo` handle as an argument.

There are several ways to get the pointer to the screen on which the window will be opened. If the application has its own custom screen, this pointer is known from the call to `OpenScreen()` or `OpenScreenTags()`. If the application already has its window opened on the Workbench or some other public screen, the screen pointer can be found in `Window.WScreen`. Often the program will create its gadgets and menus before opening the window. In this case, use `LockPubScreen()` to get a pointer to the desired public screen, which also provides a lock on the screen to prevent it from closing. See the chapters "Intuition Screens" and "Intuition Windows" for more about public screens.

The `VisualInfo` data must be freed after all the gadgets and menus have been freed but before releasing the screen. Custom screens are released by calling `CloseScreen()`, public screens are released by calling `CloseWindow()` or `UnlockPubScreen()`, depending on the technique used. Use `FreeVisualInfo()` to free the visual info data.

```
void FreeVisualInfo( APTR vi );
```

This function takes just one argument, the `VisualInfo` handle as returned by `GetVisualInfo()`. The sequence of events for using the `VisualInfo` handle could look like this:

```
init()
{
myscreen = LockPubScreen(NULL);
if (!myscreen)
{
cleanup("Failed to lock default public screen");
}
vi = GetVisualInfo(myscreen);
if (!vi)
{
cleanup("Failed to GetVisualInfo");
}
/* Create gadgets here */
ng.ng_VisualInfo = vi;
...
}
```

```

}

void cleanup(STRPTR errorstr)
{
    /* These functions may be safely called with a NULL parameter: */
    FreeGadgets(glist);
    FreeVisualInfo(vi);

    if (myscreen)
        UnlockPubScreen(NULL, myscreen);

    printf(errorstr);
}

```

1.35 15 // Setting Up GadTools Menus and Gadgets / createContext()

Use of GadTools gadgets requires some per-window context information. `CreateContext()` establishes a place for that information to go. This function must be called before any GadTools gadgets are created.

```
struct Gadget *CreateContext( struct Gadget **glistptr );
```

The `glistptr` argument is a double-pointer to a `Gadget` structure. More specifically, this is a pointer to a `NULL`-initialized pointer to a `Gadget` structure.

The return value of `CreateContext()` is a pointer to this gadget, which should be fed to the program's first call to `CreateGadget()`. This pointer to the `Gadget` structure returned by `CreateContext()`, may then serve as a handle to the list of gadgets as they are created. The code fragment listed in the next section shows how to use `CreateContext()` together with `CreateGadget()` to make a linked list of GadTools gadgets.

1.36 15 / GadTools Gadgets / Creating Gadget Lists

In the discussion of `CreateGadget()` presented earlier, the examples showed only how to make a single gadget. For most applications that use GadTools, however, a whole list of gadgets will be needed. To do this, the application could use code such as this:

```

struct NewGadget *newgad1, *newgad2, *newgad3;
struct Gadget *glist = NULL;
struct Gadget *pgad;

...
/* Initialize NewGadget structures */
...

/* Note that CreateContext() requires a POINTER to a NULL-initialized
 * pointer to struct Gadget:
 */
pgad = CreateContext(&glist);

```

```

pgad = CreateGadget (BUTTON_KIND, pgad, newgad1, TAG_END);
pgad = CreateGadget (STRING_KIND, pgad, newgad2, TAG_END);
pgad = CreateGadget (MX_KIND,      pgad, newgad3, TAG_END);

if (!pgad)
{
    FreeGadgets (glist);
    exit_error();
}
else
{
    if ( mywin=OpenWindowTags (NULL,
                               WA_Gadgets,    glist,
                               ...
                               /* Other tags... */
                               ...
                               TAG_END) )
    {
        /* Complete the rendering of the gadgets */
        GT_RefreshWindow (win, NULL);
        ...
        /* and continue on... */
        ...
        CloseWindow (mywin);
    }

    FreeGadgets (glist);
}

```

The pointer to the previous gadget, `pgad` in the code fragment above, is used for three purposes. First, when `CreateGadget()` is called multiple times, each new gadget is automatically linked to the previous gadget's `NextGadget` field, thus creating a gadget list. Second, if one of the gadget creations fails (usually due to low memory, but other causes are possible), then for the next call to `CreateGadget()`, `pgad` will be `NULL` and `CreateGadget()` will fail immediately. This means that the program can perform several successive calls to `CreateGadget()` and only have to check for failure at the end.

Finally, although this information is hidden in the implementation and not important to the application, certain calls to `CreateGadget()` actually cause several Intuition gadgets to be allocated and these are automatically linked together without program interaction, but only if a previous gadget pointer is supplied. If several gadgets are created by a single `CreateGadget()` call, they work together to provide the functionality of a single GadTools gadget. The application should always act as though the gadget pointer returned by `CreateGadget()` points to a single gadget instance. See "Documented Side-Effects" for a warning.

There is one exception to the fact that a program only has to check for failure after the last `CreateGadget()` call and that is when the application depends on the successful creation of a gadget and caches or immediately uses the gadget pointer returned by `CreateGadget()`.

For instance, if the program wants to create a string gadget and save a pointer to the string buffer, it might do so as follows:

```

gad = CreateGadget (STRING_KIND, gad, &ng,
                    GTST_String, "Hello World",
                    TAG_END);

if (gad)
{
    stringbuffer = ((struct StringInfo *) (gad->SpecialInfo))->Buffer;
}

/* Creation can continue here: */
gad = CreateGadget (..._KIND, gad, &ng2,
    ...
    TAG_END);

```

A major benefit of having a reusable NewGadget structure is that often many fields do not change and some fields change incrementally. For example, the application can set just the NewGadget's `ng_VisualInfo` and `ng_TextAttr` only once and never have to modify them again even if the structure is reused to create many gadgets. A set of similar gadgets may share size and some positional information so that code such as the following might be used:

```

/* Assume that the NewGadget structure 'ng' is fully
 * initialized here for a button labelled "OK"
 */
gad = CreateGadget (BUTTON_KIND, gad, &ng,
    TAG_END);

/* Modify only those fields that need to change: */
ng.ng_GadgetID++;
ng.ng_LeftEdge += 80;
ng.ng_GadgetText = "Cancel";
gad = CreateGadget (BUTTON_KIND, gad, &ng,
    TAG_END);

```

Warning:

 All gadgets created by GadTools currently have the `GADTOOL_TYPE` bit set in their `GadgetType` field. It is not correct to check for, set, clear or otherwise rely on this since it is subject to change.

1.37 15 / GadTools Gadgets / Gadget Refresh Functions

Normally, GadTools gadgets are created and then attached to a window when the window is opened, either through the `WA_Gadget` tag or the `NewWindow.FirstGadget` field. Alternately, they may be added to a window after it is open by using the functions `AddGLList()` and `RefreshGLList()`.

Regardless of which way gadgets are attached to a window, the program must then call the `GT_RefreshWindow()` function to complete the rendering of GadTools gadgets. This function takes two arguments.

```
void GT_RefreshWindow( struct Window *win, struct Requester *req );
```

This `win` argument is a pointer to the window that contains the GadTools

gadgets. The req argument is currently unused and should be set to NULL. This function should only be called immediately after adding GadTools gadgets to a window. Subsequent changes to GadTools gadget imagery made through calls to GT_SetGadgetAttrs() will be automatically performed by GadTools when the changes are made. (There is no need to call GT_RefreshWindow() in that case.)

As mentioned earlier, applications must always ask for notification of window refresh events for any window that uses GadTools gadgets. When the application receives an IDCMP_REFRESHWINDOW message for a window, Intuition has already refreshed its gadgets. Normally, a program would then call Intuition's BeginRefresh(), perform its own custom rendering operations, and finally call EndRefresh(). But for a window that uses GadTools gadgets, the application must call GT_BeginRefresh() and GT_EndRefresh() in place of BeginRefresh() and EndRefresh(). This allows the the GadTools gadgets to be fully refreshed.

```
void GT_BeginRefresh( struct Window *win );
void GT_EndRefresh ( struct Window *win, long complete );
```

For both functions, the win argument is a pointer to the window to be refreshed. For GT_EndRefresh(), set the complete argument to TRUE if refreshing is complete, set it to FALSE otherwise. See the discussion of BeginRefresh() and EndRefresh() in the "Intuition Windows" chapter for more about window refreshing.

When using GadTools gadgets, the program may not set the window's WFLG_NOCAREREFRESH flag. Even if there is no custom rendering to be performed, GadTools gadgets requires this minimum code to handle IDCMP_REFRESHWINDOW messages:

```
case IDCMP_REFRESHWINDOW:
    GT_BeginRefresh(win);
    /* custom rendering, if any, goes here */
    GT_EndRefresh(win, TRUE);
    break;
```

1.38 15 / GadTools Gadgets / Other GadTools Functions

This section discusses some additional support functions in the GadTools library that serve special needs.

GT_FilterIMsg() and GT_PostFilterIMsg() DrawBevelBox()

1.39 15 // Other Functions / GT_FilterIMsg() and GT_PostFilterIMsg()

For most GadTools programs, GT_GetIMsg() and GT_ReplyIMsg() work perfectly well. In rare cases an application may find they pose a bit of a problem. A typical case is when all messages are supposed to go through a centralized ReplyMsg() that cannot be converted to a GT_ReplyIMsg(). Since calls to GT_GetIMsg() and GT_ReplyIMsg() must be paired, there would be a problem.

For such cases, the `GT_FilterIMsg()` and `GT_PostFilterIMsg()` functions are available. These functions allow `GetMsg()` and `ReplyMsg()` to be used in a way that is compatible with GadTools.

Warning:

These functions are for specialized use only and will not be used by the majority of applications. See `GT_GetIMsg()` and `GT_ReplyIMsg()` for standard message handling.

```
struct IntuiMessage *GT_FilterIMsg( struct IntuiMessage *imsg );
struct IntuiMessage *GT_PostFilterIMsg( struct IntuiMessage *imsg );
```

The `GT_FilterIMsg()` function should be called right after `GetMsg()`. It takes a pointer to the original `IntuiMessage` and, if the message applies to a GadTools gadget, returns either a modified `IntuiMessage` or a `NULL`. A `NULL` return signifies that the message was consumed by a GadTools gadget (and not needed by the application).

The `GT_PostFilterIMsg()` function should be called before replying to any message modified by `GT_FilterIMsg()`. It takes a pointer to the modified version of an `IntuiMessage` obtained with `GT_FilterIMsg()` and returns a pointer to the original `IntuiMessage`.

The typical calling sequence for a program that uses these functions, is to call `GetMsg()` to get the `IntuiMessage`. Then, if the message applies to a window which contains GadTools gadgets, call `GT_FilterIMsg()`. Any message returned by `GT_FilterIMsg()` should be used like a message returned from `GT_GetIMsg()`.

When done with the message, the application must call `GT_PostFilterIMsg()` to perform any clean up necessitated by the previous call to `GT_FilterIMsg()`. In all cases, the application must then reply the original `IntuiMessage` using `ReplyMsg()`. This is true even for consumed messages as these are not replied by GadTools. For example, the application could use code such as this:

```
/* port is a message port receiving different messages */
/* gtwindow is the window that contains GadTools gadgets */

imsg = GetMsg(port);

/* Is this the window with GadTools gadgets? */
if (imsg->IDCMPWindow == gtwindow)
{
    /* Filter the message and see if action is needed */
    if (gtimsg = GT_FilterIMsg(imsg))
    {
        switch (gtimsg->Class)
        {
            {
                /* Act depending on the message */
                ...
            }
        }
        /* Clean up the filtered message. The return value is not */
        /* needed since we already have a pointer to the original */
    }
}
```

```

        /* message.                                     */
        GT_PostFilterIMsg(gtimg);
    }
    /* other stuff can go here */
    ReplyMsg(img);

```

You should not make any assumptions about the contents of the unfiltered `IntuiMessage` (`img` in the above example). Only two things are guaranteed: the unfiltered `IntuiMessage` must be replied to and the unfiltered `IntuiMessage` (if it produces anything when passed through `GT_FilterIMsg()`) will produce a meaningful GadTools `IntuiMessage` like those described in the section on the different kinds of gadgets. The relationship between the unfiltered and filtered messages are expected to change in the future. See the section on documented side-effects for more information.

1.40 15 // Other GadTools Functions / DrawBevelBox()

A key visual signature shared by most GadTools gadgets is the raised or recessed bevelled box imagery. Since the program may wish to create its own boxes to match, GadTools provides the `DrawBevelBox()` and `DrawBevelBoxA()` functions.

```

void DrawBevelBoxA( struct RastPort *rport, long left, long top,
                    long width, long height, struct TagItem *taglist );
void DrawBevelBox ( struct RastPort *rport, long left, long top,
                    long width, long height, Tag tag1, ... );

```

The `rport` argument is a pointer to the `RastPort` into which the box is to be rendered. The `left`, `top`, `width` and `height` arguments specify the dimensions of the desired box.

The tag arguments, `tag1` or `taglist`, may be set as follows:

`GT_VisualInfo` (APTR)

The `VisualInfo` handle as returned by a prior call to `GetVisualInfo()`. This value is required.

`GTBB_Recessed` (BOOL)

A bevelled box may either appear to be raised to signify an area of the window that is selectable or recessed to signify an area of the window in which clicking will have no effect. Set this boolean tag to `TRUE` to get a recessed box. Omit this tag entirely to get a raised box.

`DrawBevelBox()` is a rendering operation, not a gadget. This means that the program must refresh any bevelled boxes rendered through this function if the window gets damaged.

1.41 15 / GadTools Gadgets / Gadget Keyboard Equivalents

Often, users find it convenient to control gadgets using the keyboard. Starting with V37, it is possible to denote the keyboard equivalent for a GadTools gadget. The keyboard equivalent will be an underscored character in the gadget label, for easy identification. At the present time, however, the application is still responsible for implementing the reaction to each keypress.

Denoting a Gadget's Keyboard Equivalent
Implementing a Gadget's Keyboard Equivalent Behavior

1.42 15 // Keyboard Equivalents / Denoting a Gadget's Keyboard Equivalent

In order to denote the key equivalent, the application may add a marker-symbol to the gadget label. This is done by placing the marker-symbol immediately before the character to be underscored. This symbol can be any character that is not used in the label. The underscore character, `'_'` is the recommended marker-symbol. So, for example, to mark the letter "F" as the keyboard equivalent for a button labelled "Select Font...", create the gadget text:

```
ng.ng_GadgetText = "Select _Font...";
```

To inform GadTools of the underscore in the label, pass the `GA_Underscore` tag to `CreateGadget()` or `CreateGadgetA()`. The data-value associated with this tag is a character, not a string, which is the marker-symbol used in the gadget label:

```
GA_Underscore, '_' /* Note '_', not "_" !!! */
```

GadTools will create a gadget label which consists of the text supplied with the marker-symbol removed and the character following the marker-symbol underscored.

The gadget's label would look something like:

```
Select Font...  
-
```

1.43 15 /// Implementing a Gadget's Keyboard Equivalent Behavior

Currently, GadTools does not process keyboard equivalents for gadgets. It is up to the application writer to implement the correct behavior, normally by calling `GT_SetGadgetAttrs()` on the appropriate gadget. For some kinds of gadget, the behavior should be the same regardless of whether the keyboard equivalent was pressed with or without the shift key. For other gadgets, shifted and unshifted keystrokes will have different, usually opposite, effects.

Here is the correct behavior for keyboard equivalents for each kind of GadTools gadget:

Button Gadgets

The keyboard equivalent should invoke the same function that clicking on the gadget does. There is currently no way to highlight the button visuals programmatically when accessing the button through a keyboard equivalent.

Text-Entry and Number-Entry Gadgets

The keyboard equivalent should activate the gadget so the user may type into it. Use Intuition's `ActivateGadget()` call.

Checkbox Gadgets

The keyboard equivalent should toggle the state of the checkbox. Use `GT_SetGadgetAttrs()` and the `GTCB_Checked` tag.

Mutually-Exclusive Gadgets

The unshifted keystroke should activate the next choice, wrapping around from the last to the first. The shifted keystroke should activate the previous choice, wrapping around from the first to the last. Use `GT_SetGadgetAttrs()` and the `GTMX_Active` tag.

Cycle Gadgets

The unshifted keystroke should activate the next choice, wrapping around from the last to the first. The shifted keystroke should activate the previous choice, wrapping around from the first to the last. Use `GT_SetGadgetAttrs()` and the `GTCY_Active` tag.

Slider Gadgets

The unshifted keystroke should increase the slider's level by one, stopping at the maximum, while the shifted keystroke should decrease the level by one, stopping at the minimum. Use `GT_SetGadgetAttrs()` and the `GTSL_Level` tag.

Scroller Gadgets

The unshifted keystroke should increase the scroller's top by one, stopping at the maximum, while the shifted keystroke should decrease the scroller's top by one, stopping at the minimum. Use `GT_SetGadgetAttrs()` and the `GTSC_Top` tag.

Listview Gadgets

The unshifted keystroke should cause the next entry in the list to become the selected one, stopping at the last entry, while the shifted keystroke should cause the previous entry in the list to become the selected one, stopping at the first entry. Use `GT_SetGadgetAttrs()` and the `GTLV_Top` and `GTLV_Selected` tags.

Palette Gadgets

The unshifted keystroke should select the next color, wrapping around from the last to the first. The shifted keystroke should activate the previous color, wrapping around from the first to the last. Use `GT_SetGadgetAttrs()` and the `GTPA_Color` tag.

Text-Display and Number-Display Gadgets

These kinds of GadTools gadget have no keyboard equivalents since they are not selectable.

Generic Gadgets

Define appropriate keyboard functions based on the kinds of keyboard

behavior defined for other GadTools kinds.

1.44 15 / GadTools Gadgets / Restrictions on GadTools Gadgets

There is a strict set of functions and operations that are permitted on GadTools gadgets. Even if a technique is discovered that works for a particular case, be warned that it cannot be guaranteed and should not be used. If the trick concocted only works most of the time, it may introduce subtle problems in the future.

Never selectively or forcibly refresh gadgets. The only gadget refresh that should ever be performed is the initial `GT_RefreshWindow()` after a window is opened with GadTools gadgets attached. It is also possible to add gadgets after the window is opened by calling `AddGlist()` and `RefreshGlist()` followed by `GT_RefreshWindow()`. These refresh functions should not be called at any other time.

GadTools gadgets may not overlap with each other, with other gadgets or with other imagery. Doing this to modify the gadget's appearance is not supported.

GadTools gadgets may not be selectively added or removed from a window. This has to do with the number of Intuition gadgets that each call to `CreateGadget()` produces and with refresh constraints.

Never use `OnGadget()` or `OffGadget()` or directly modify the `GFLG_DISABLED` Flags bit. The only approved way to disable or enable a gadget is to use `GT_SetGadgetAttrs()` and the `GA_Disabled` tag. Those kinds of GadTools gadgets that do not support `GA_Disabled` may not be disabled (for now).

The application should never write into any of the fields of the Gadget structure or any of the structures that hang off it, with the exception noted earlier for `GENERIC_KIND` gadgets. Avoid making assumptions about the contents of these fields unless they are explicitly programmer fields (`GadgetID` and `UserData`, for example) or if they are guaranteed meaningful (`LeftEdge`, `TopEdge`, `Width`, `Height`, `Flags`). On occasion, the program is specifically invited to read a field, for example the `StringInfo->Buffer` field.

GadTools gadgets may not be made relative sized or relative positioned. This means that the gadget flags `GFLG_RELWIDTH`, `GFLG_RELHEIGHT`, `GFLG_RELBOTTOM` and `GFLG_RELRIGHT` may not be specified. The activation type of the gadget may not be modified (for example changing `GACT_IMMEDIATE` to `GACT_RELVERIFY`). The imagery or the highlighting method may not be changed.

These restrictions are not imposed without reason; subtle or blatant problems may arise now or in future versions of GadTools for programs that violate these guidelines.

1.45 15 / GadTools Gadgets / Documented Side-Effects

There are certain aspects of the behavior of GadTools gadgets that should not be depended on. This will help current applications remain compatible with future releases of the GadTools library.

When using `GT_FilterIMsg()` and `GT_PostFilterIMsg()`, never make assumptions about the message before or after filtering. I.e., do not interpret the unfiltered message, assume that it will or will not result in certain kinds of filtered message or assume it will result in a consumed message (i.e., when `GT_FilterIMsg()` returns `NULL`).

`IDCMP_INTUITICKS` messages are consumed when a scroller's arrows are repeating. That is, `IDCMP_INTUITICKS` will not be received while the user is pressing a scroller arrows. Do not depend or rely on this side effect, though, it will not necessarily remain so in the future.

A single call to `CreateGadget()` may create one or more actual gadgets. These gadgets, along with the corresponding code in GadTools, define the behavior of the particular kind of GadTools gadget requested. Only the behavior of these gadgets is documented, the number or type of actual gadgets is subject to change. Always refer to the gadget pointer received from `CreateGadget()` when calling `GT_SetGadgetAttrs()`. Never refer to other gadgets created by GadTools, nor create code which depends on their number or form.

For text-display gadgets, the `GTTX_CopyText` tag does not cause the text to be copied when the text is later changed with `GTTX_Text`.

The `PLACETEXT ng_Flags` are currently ignored by `GENERIC_KIND` gadgets. However, this may not always be so.

All GadTools gadgets set `GADTOOL_TYPE` in the gadget's `GadgetType` field. Do not use this flag to identify GadTools gadgets, as this flag is not guaranteed to be set in the future.

The palette gadget subdivides its total area into the individual color squares. Do not assume that the subdivision algorithm won't change.

1.46 15 GadTools Library / Function Reference

The following are brief descriptions of the Intuition functions discussed in this chapter. See the "Amiga ROM Kernel Reference Manual: Includes and Autodocs" for details on each function call. All of these functions require Release 2 or a later version of the operating system.

Table 15-2: GadTools Library Functions

Function	Description
<code>CreateGadgetA()</code>	Allocate GadTools gadget, tag array form.
<code>CreateGadget()</code>	Allocate GadTools gadget, varargs form.
<code>FreeGadgets()</code>	Free all GadTools gadgets.
<code>GT_SetGadgetAttrsA()</code>	Update gadget, tag array form.

GT_SetGadgetAttrs()	Update gadget, varargs form.
CreateContext()	Create a base for adding GadTools gadgets.

CreateMenusA()	Allocate GadTools menu structures, tag array form.
CreateMenus()	Allocate GadTools menu structures, varargs form.
FreeMenus()	Free menus allocated with CreateMenus().
LayoutMenuItemsA()	Format GadTools menu items, tag array form.
LayoutMenuItems()	Format GadTools menu items, varargs form.
LayoutMenusA()	Format GadTools menus, tag array form.
LayoutMenus()	Format GadTools menus, varargs form.

GT_GetIMsg()	GadTools gadget compatible version of GetMsg().
GT_ReplyIMsg()	GadTools gadget compatible version of ReplyMsg().
GT_FilterIMsg()	Process GadTools gadgets with GetMsg()/ReplyMsg().
GT_PostFilterIMsg()	Process GadTools gadgets with GetMsg()/ReplyMsg().

GT_RefreshWindow()	Display GadTools gadget imagery after creation.
GT_BeginRefresh()	GadTools gadget compatible version of BeginRefresh().
GT_EndRefresh()	GadTools gadget compatible version of EndRefresh().

DrawBevelBoxA()	Draw a 3D box, tag array form.
DrawBevelBox()	Draw a 3D box, varargs form.

GetVisualInfoA()	Get drawing information for GadTools, tag array form.
GetVisualInfo()	Get drawing information for GadTools, varargs form.
FreeVisualInfo()	Free drawing information for GadTools.