GADOUTLINE LIBRARY

The *gadoutline.library* contains a set of functions designed to assist Amiga programmers in the design and implementation of an application's GUI (Graphical User Interface). In order to achieve this goal, it provides three primary services to the programmer:

- A method of specifying an application's GUI layout that is independent of the font it is using and the size of the window it is in. This takes care of most of the work a program must do in order to respect the user's preferred font and support resizable windows.

- Automatic allocation, tracking and deallocation of all objects used by the GUI. This is available for both GadTools and BOOPSI gadgets and includes built-in support for hot-keys and window resizing.

- Additional support functions for such things as memory tracking, requesters, IDCMP message processing, sharing message ports with multiple windows and simple vector drawings.

This library is built around the standard AmigaOS gadtools.library and intuition.library. If you are unfamiliar with these topics, it is recommended that you refer to Amiga ROM Kernel Reference Manual: Libraries, chapters 4, 9 & 15, for more information. In addition, basic knowledge of tag lists, as described in chapter 37, is assumed.

The Outline Array

One of the fundamental data structures used by gadoutline.library is called the *outline array*. It is used by the application to describe its user interface for the library. Essentially, it is a large array of 32-bit ULONGs, organized as a series of variable-length *commands* instructing the library on how to create your interface. Each of these commands has three unique components: they always begin with a single ULONG *identifier*, followed by 0 to 8 *parameter* ULONGs and ending with 0 to 3 *tag lists*. The identifier contains not only information about what the command is, but also the number of parameters and tag lists which follow.

A command must be followed by the exact number of parameters and tag lists that its identifier specifies. These are the only pieces of information used by the library to determine the location of the next command; the part of the identifier describing what a command actually does only determines how the parameters and tag lists are interpreted, and not how many there are.

OUTLINE ARRAY STRUCTURE

While the structure an outline command is fairly complex, a number of C-language macros are supplied to assist in its creation. When used in an outline, these macros look like function calls with parameters, followed by any tag lists the command needs:

```
ULONG OutlineArray[] = {
    GO_GTBOX(TEXT_KIND,0,1,0,(ULONG)&"My Text:",PLACETEXT_LEFT),
        GOCT_SetHotKey, 't',
        TAG_END,
        GTTX_Text, (ULONG)&"This is some text",
        GTTX_Border, TRUE,
        GT_Underscore, '_',
        TAG_END
    GO_ENDOUTLINE()
};
```

This will create a **TEXT_KIND** GadTools gadget. There are a number of important points which this simple example brings out:

- The outline array is declared as an array of ULONGs within your program.
- Text directly included in the array must be referenced as (ULONG)&"text". This means "give me the address of this string, and make it into a ULONG value."
- The command **GO_GTBOX** has 6 parameters - the GadTools *Kind*, *Group ID*, *Standard ID*, *Weight*, *Label text* and *Label position*.
- The command also includes two tag lists. The first is a list of command tags for the library, in this case being used to assign the *hot key* 't' to this command's gadget. The second is a list of standard tags as used by GadTools.
- Every outline array must include the command **GO_ENDOUTLINE()**, which tells the library that it has reached the end of the array.

TAG LISTS IN AN OUTLINE ARRAY

An unusual feature of the outline array is its use of 'imbedded' tag lists. These are parsed as normal tag lists by the library, except that the TAG_END is always used to mark the end of the tag list, and thus is immediately followed by the next tag list or command identifier in the array. This means that all imbedded tag lists must be terminated with a TAG_END, even if it contains a TAG_MORE, with the TAG_END then being a delimitator between individual commands and tag lists:

Incorrect: (a '0' for ti_Data should not be placed after TAG_END.)

```
..., GT_Underscore, '_', TAG_END, 0, GO_GTBOX(...
```

Correct:

```
..., GT_Underscore, '_', TAG_END, GO_GTBOX(...
```

Incorrect: (TAG_MORE does not mark the end of an imbedded tag list.)

```
..., GT_Underscore, '_', TAG_MORE, (ULONG)&AdditionalTags[0], GO_GTBOX(...
..., GOCT_SetHotKey, 't', TAG_MORE, (ULONG)&AdditionalTags[0], GT_Underscore, ...
```

Correct:

```
..., GT_Underscore, '_', TAG_MORE, (ULONG)&AdditionalTags[0], TAG_END, GO_GTBOX(...
..., GOCT_SetHotKey, 't', TAG_MORE, (ULONG)&AdditionalTags[0], TAG_END, GT_Underscore, ...
```

Outline Commands

The outline commands are the main source of interaction between the application and the library. Not only are they used to describe a GUI's layout, but they are also the point of interface between the application and the graphical objects which the library creates. There are four important components to a command: An *ID code* which is used by the application to refer to the command; the command's *tag lists* supply the command with more detailed information about its desired behavior; the *relationship between commands* within the outline is used to determine where gadgets should actually be positioned in the display; and the *type* of a command determines its overall function within the outline.

OUTLINE COMMAND ID CODES

Because the library creates and tracks a command's object (for example, a ListView gadget) internally, the application is unable to directly access these objects with pointers. Instead, the programmer must assign a unique ID code, called the *Standard ID*, to the command which created the object, and then use this ID code to indirectly reference the object. This code ranges from 1 to 4095, while a special ID code 0 (zero), meaning *'no code*,' may be assigned to all commands which the application does not need to reference. For example, the library function **GO_SetObjAttrs()**, which is used to set a command's object attributes much like GadTools' **GT_SetGadgetAttrs()**, takes a command's ID number as a parameter rather than the gadget pointer which GT_SetGadgetAttrs() requires:

```
void GO_SetObjAttrs(struct GadOutline *GadOutline, CMDID CmdID, ULONG Flags,
                    ULONG Tag1Type, ...);

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

In addition, every command within an outline may have another ID code, called the *Group ID*. In contrast to the *Standard ID*, the Group ID does not need to be unique to a single command, and in fact is mainly used to associate a group of similar commands together. For example, you may have a set of gadgets which all need to be disabled at certain points in your program. If you assign all of them the same Group ID, you can call **GO_SetObjGrpAttrs()** with this Group ID to automatically disable all of the commands which match the ID.

The Group ID is further divided into two more components: The *Group ID Mask* is a set of 12 bits which may be set on or off for each command, while the *Group ID Code* is a single 1 to 255 value (or, as with the Standard ID, a value of 0 for '*no code*') which may be assigned to each command. A special macro is supplied to create an ID code. **GO_CMDID(GrpID, StdID)** takes two arguments, the Group ID and Standard ID of the command, and creates a 32-bit value result. This macro is only necessary if you are using a Group ID; otherwise, you may just directly supply the Standard ID you want and leave the group bits zero.

When operating on a set of commands, all three ID codes - the *Standard ID*, *Group ID Mask* and *Group ID Code* must match the requested ID code or be zero. (In this case, each bit of the group mask is treated as a separate component of the ID code.) For example, assuming you have seven commands in your Outline array assigned the codes:

Cmd1: `GO_CMDID( 0, 1 )`
Cmd2: `GO_CMDID( GO_GRPID_A, 2 )`
Cmd3: `GO_CMDID( GO_GRPID_A | GO_GRPID_B, 3 )`
Cmd4: `GO_CMDID( GO_GRPID_A | 100, 4 )`
Cmd5: `GO_CMDID( GO_GRPID_B | 100, 5 )`
Cmd6: `GO_CMDID( GO_GRPID_A | GO_GRPID_B | 100, 6 )`
Cmd7: `GO_CMDID( 0, 0 )`

A request for a Command ID of 1 will return only Cmd1; an ID of 2 only Cmd2, etc.
An ID of `GO_CMDID(GO_GRPID_B,0)` will return Cmd3, Cmd5, Cmd6.
`GO_CMDID(GO_GRPID_A,0)` will return Cmd2, Cmd3, Cmd4, Cmd6.
`GO_CMDID(GO_GRPID_A|100,0)` will return Cmd4 and Cmd6.
`GO_CMDID(0,0)` will return all commands, including Cmd7.

OUTLINE COMMAND TAGS

The first tag list supplied to every command is a set of tags called the *Command Tag List*. These are tags which are defined by the library and are used to control the general behavior of commands; they can be identified by the text **GOCT_** prepended to them. This behavior includes such things as specifying a *hot key* attached to a command, *linking* commands together (for example, attaching a ListView to a string gadget), or adjusting the layout of the display. In addition, the second tag list of a command is designated as the *Object Tag List*. These are tags which are specific for the object which the command is creating - for example, a command which creates a GadTools gadget passes its object

tags directly to **CreateGadget()** without interpreting them. Similarly, a BOOPSI command will pass these same tags to the function **NewObject()**.

Some of the more common command tags are:

> **GOCT_SetHotKey**: Attach a *hot key* to this command. The tag's **ti_Data** is the ASCII character code of the hot key to attach to the command. All **v37** GadTools kinds perform appropriate actions when they receive a hot key event. For example, string gadgets will activate themselves and ListView gadgets will move through their list. In addition, a special ti_Data value of 0 (zero) will request that the library assign a hot key for you. This is supported for the GadTools gadgets by determining the character underlined in the gadget's label text, and assigning this as the hot key.
>
> **GOCT_LinkFromTag** and **GOCT_LinkToStdID**: These two tags are used to link objects together. They must occur in the tag list together, in the order shown. **GOCT_LinkFromTag** specifies the *tag item* within this command's object tag list which links it to another command, while **GOCT_LinkToStdID** supplies the *Standard ID* code of the command to which this tag will be linked. For example, the GadTools tag **GTLV_ShowSelected** requires a pointer to a string gadget; because we do not know what this pointer will be when creating the outline array, we will need to create a link from the ListView command to the command which creates the string gadget. Assuming the string gadget is assigned an ID of 2, then:
>
> ```
> ..., GOCT_LinkFromTag, GTLV_ShowSelected, GOCT_LinkToStdID, 2, ...
> ```
>
> placed in the ListView's command tag list will instruct the library to fill in the ListView's GTLV_ShowSelected tag with the appropriate pointer to the string.

OUTLINE OBJECT LAYOUT

The actual positioning of objects is based on a recursive organization of *boxes* and *groups*. A box is the simplest display element, and basically contains one display 'object'. This may be, for example, a ListView gadget, a drawing, or anything else which takes up a non-divisible block of area on the display. A group can be thought of as a 'container' for other boxes or groups. It may have either a horizontal or vertical orientation, and organizes the boxes and groups within it across this orientation, in the order in which they appear in the outline. Because any group may contain additional groups within it, complex layout descriptions may be created which allow the library to determine exactly where each box should ultimately be placed with respect to all of the others.

A simple layout, containing a horizontal and a vertical group, may look like Figure 1.

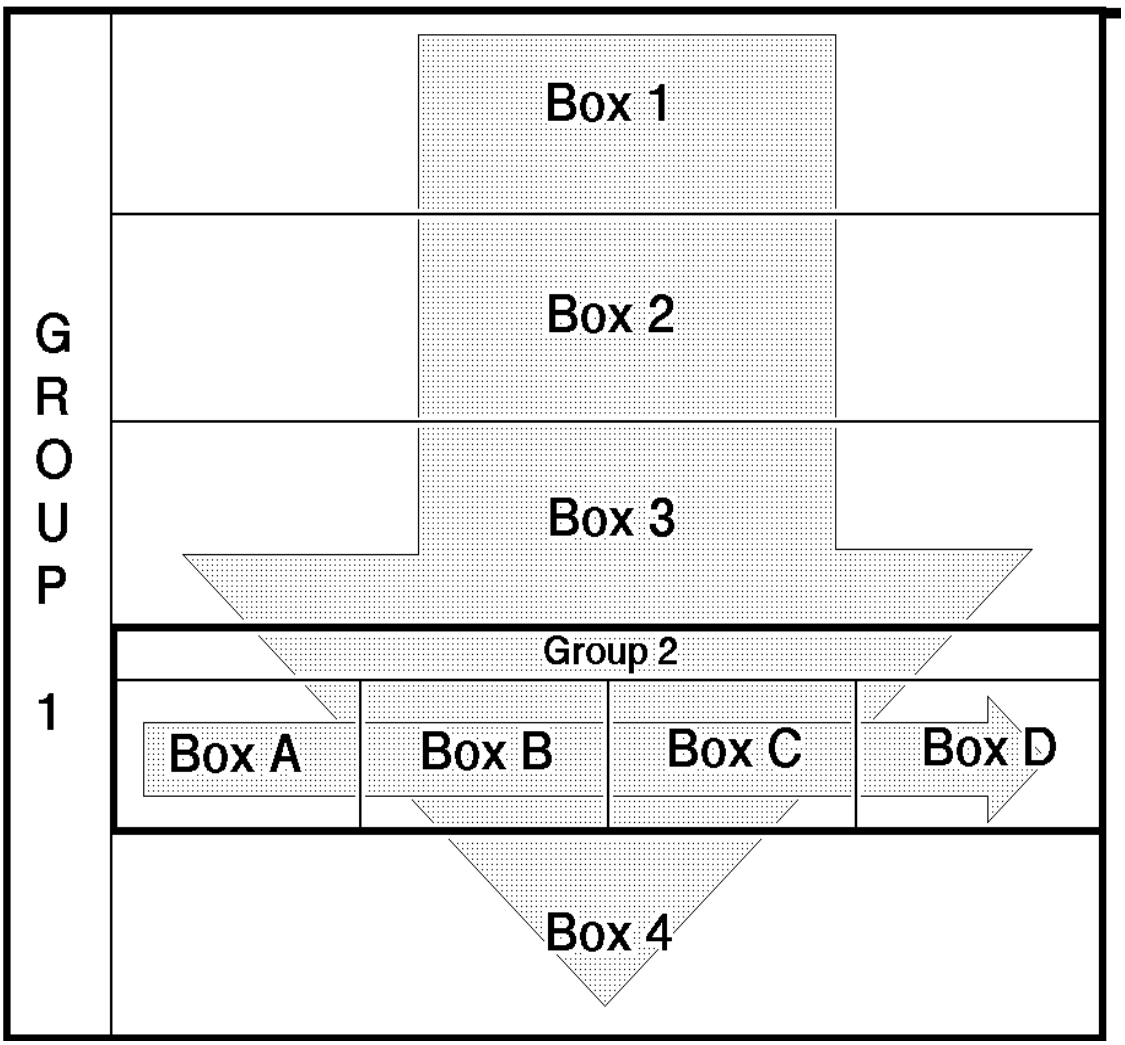

Figure 1: Simple Group and Box layout.

This layout is comprised of a vertical group (Group 1), containing three boxes (Box 1, Box 2, Box3) followed by a horizontal group (Group 2) and another box (Box 4). The horizontal group itself then contains another four boxes (Box A, Box B, Box C, Box D). The basic structure of the outline array used to create this layout would then be the following:

```
Create Vertical Group (ID = Group 1).
  Create Box (ID = Box 1).
  Create Box (ID = Box 2).
  Create Box (ID = Box 3).
  Create Horizontal Group (ID = Group 2).
       Create Box (ID = Box A).
       Create Box (ID = Box B).
       Create Box (ID = Box C).
       Create Box (ID = Box D).
  Create Box (ID = Box 4).
```

A number of factors come into play when determining the exact position of objects within a group. Every box has a "*minimum size*;" this is the smallest size that the box is allowed to be. A group then finds its minimum size by looking at the minimum sizes of the objects within it. For example, in a vertical group, the minimum height would be the sum of the heights of the all boxes within it, and the minimum width would be the maximum of all the boxes' widths.

In addition, every group and box is assigned a *weight*, which allows the library to determine how to distribute any extra space within a group which is not taken up by its object's minimum dimensions. This weight is an arbitrary number assigned to each object, and defines the amount of space to give to that object relative to all of the others within the group. For example, imagine a group which contains two boxes, of weight `1` and `2` respectively, and which has `100` extra pixels it needs to distribute to reach its desired dimension. The total weight of all of the boxes within the group is `(1+2)` or `3`; so the first box will be assigned `(1/3)*100 = 33` pixels and the second with receive `(2/3)*100 = 66` pixels. Note that the amount of extra space assigned to a box is in addition to its minimum size.

OUTLINE COMMAND CLASSES

There are a number of commands which may be used in an outline, although they can be grouped into four main classes: object creation commands, layout control commands, outline control commands and global settings commands.

OBJECT CREATION COMMANDS

These commands are used to create the various components of an application's GUI. This includes the commands **GO_GTBOX()**, **GO_DRAWBOX()**, **GO_BOOPSIBOX()** and **GO_EMPTYBOX()**. These are used to create single object elements within the display - for example, a string gadget, a label or a ListView.

GO_GTBOX(Kind,GrpID,StdID,Weight,Name,Flags): Creates a GadTools gadget. The command's object tag list should contain standard GadTools tags.

Kind: The GadTools kind; for example, LISTVIEW_KIND. See the GadTools documentation for appropriate values.

GrpID: *Group ID* code to assign to the command.

StdID: *Standard ID* code to assign to the command and gadget.

Weight: This command's relative weight; see *OUTLINE OBJECT LAYOUT* above for details on the meaning of this field.

Name: GadTools gadget label.

Flags: GadTools flags, i.e. PLACETEXT_LEFT.

GO_DRAWBOX(Type,GrpID,StdID,Weight): Creates a GadOutline drawing. The command's object tag list should contain GadOutline drawing tags - i.e., tags prepended with the code **GODT_**.

Type: Type of drawing to create. May be one of:
- **GOSD_Normal**: A normal drawing.
- **GOSD_Button**: Drawing with a GadTools button placed on it which inverts the image when it is pressed.
- **GOSD_BoopsiIm**: Drawing encapsulated inside a BOOPSI image object. This image may be linked to another BOOPSI object using a TagLink.
- **GOSD_BoopsiGad**: Drawing encapsulated inside a complete BOOPSI button gadget, which supports disabling and intelligent inverting of the image by swapping the drawing's pens.

GrpID: Group ID code to assign to the command.

StdID: Standard ID code to assign to the command (and gadget, if created).

Weight: This command's relative weight; see *OUTLINE OBJECT LAYOUT* above for details on the meaning of this field.

GO_BOOPSIBOX(Type,GrpID,StdID,Weight,ClassName,ClassID): Creates a BOOPSI gadget. The command's object tag list should contain standard BOOPSI tags appropriate to the type of BOOPSI gadget being created.

Type: Type of BOOPSI gadget to create; this must match the object type which is actually created, and is used by the library to determine how to handle the object within the display. May be one of:
- **GOSB_Object**: Generic object; for example, a 'modelclass'.
- **GOSB_Image**: BOOPSI image; for example, a 'frameiclass'.
- **GOSB_Gadget**: BOOPSI gadget; for example, a 'propgclass'.
- **GOSB_AddGad**: BOOPSI gadget as above, which is added to the list of BOOPSI gadgets which will be attached to the window's gadget list.

GrpID: Group ID code to assign to the command.

StdID: Standard ID code to assign to the command and gadget.

Weight: This command's relative weight; see *OUTLINE OBJECT LAYOUT* above for details on the meaning of this field.

ClassName: Public name of the BOOPSI object class to be created.

ClassID: Address to a private BOOPSI object class to create. These last two parameters correspond exactly to the parameters passed to **NewObject()**. See the Intuition BOOPSI documentation for details on their use.

GO_EMPTYBOX(GrpID,StdID,Weight): Creates an empty box. This is useful for creating spacing between boxes.

GrpID: Group ID code to assign to the command.

StdID: Standard ID code to assign to the command.

Weight: This command's relative weight; see *OUTLINE OBJECT LAYOUT* above for details on the meaning of this field.

LAYOUT CONTROL COMMANDS

These are the vertical and horizontal grouping commands used to create the 'structural' organization of the outline, which the library uses to determine where to place the gadgets in the display. Note that every group must be terminated with the **GO_ENDGRP()** command, as described in the next section.

GO_HORIZGRP(GrpID,StdID,Weight): Creates a horizontally-oriented group; objects within this group run from left to right.

GrpID: Group ID code to assign to the command.
StdID: Standard ID code to assign to the command.
Weight: This command's relative weight; see *OUTLINE OBJECT LAYOUT* above for details on the meaning of this field.

GO_VERTGRP(GrpID,StdID,Weight): Creates a vertically-oriented group; objects within this group run from top to bottom.

GrpID: Group ID code to assign to the command.
StdID: Standard ID code to assign to the command.
Weight: This command's relative weight; see *OUTLINE OBJECT LAYOUT* above for details on the meaning of this field.

OUTLINE CONTROL COMMANDS

These commands are used to structure the parsing of the outline; they are used to mark the end of previously declared sections of the outline array.

GO_ENDGRP(): End a group definition. This causes the layout engine to pop back into the group which contains the current one. Every group command (**GO_HORIZGRP()** and **GO_VERTGRP()**) must have an associated GO_ENDGRP() command.

GO_ENDOUTLINE(): End the outline array. This marks the end of the commands; the library will immediately stop parsing commands at this point. EVERY OUTLINE MUST END WITH THIS COMMAND.

GLOBAL SETTINGS COMMANDS

These commands are used to control various global settings and options in the outline. They affect the settings at the point at which they occur in the outline; any commands preceding them will see the previous settings, while any commands following them will get the new ones.

GO_OUTLINETAGS(GrpID,StdID): Allows the standard GadOutline global tags (tags starting with **GOA_**) to be modified. These are the tags which are passed to the function AllocGadOutline() when an outline is initially created. The object (second) tag list contains the outline tags.
GrpID: Group ID code to assign to the command.
StdID: Standard ID code to assign to the command.

GO_WINDOWTAGS(GrpID,StdID): Allows window tags, as passed to Intuition's **OpenWindowTags()** function, to be included in the outline. When **GO_OpenWindow()** is called, these tags will be included with any directly supplied to the function. The object (second) tag list contains the window tags.
GrpID: Group ID code to assign to the command.
StdID: Standard ID code to assign to the command.

Example Outline Array

The following is an example definition of an outline array. It creates a layout with six GadTools buttons, arranged in two rows of three columns.

```
// Define the Standard IDs which will be assigned to the commands.

enum {
    GAD1_ID = 1,
    GAD2_ID,
    GAD3_ID,
    GAD4_ID,
    GAD5_ID,
    GAD6_ID
};

// Start the Outline Array.

ULONG Outline[] = {

// Default GadOutline GOA_ type tags.  These tags are immediately parsed
// when this command is encountered just as if they were passed to
// AllocGadOutline().

GO_OUTLINETAGS(0,0),

    TAG_END,                        // 1st tag list -- no command tags.
    GOA_BaseName,       (ULONG)&"GOArrayExample",
    TAG_END,

// Default window tags.  These are supplied as defaults when GO_OpenWindow()
// is called; they will be overridden by any tags passed directly to the
// function call.  Only include this command once.

GO_WINDOWTAGS(0,0),

    TAG_END,                        // 1st tag list -- no command tags.
    WA_IDCMP,           IDCMP_CLOSEWINDOW | IDCMP_REFRESHWINDOW | IDCMP_NEWSIZE,
    WA_Activate,        TRUE,
    WA_CloseGadget,     TRUE,
    WA_DepthGadget,     TRUE,
    WA_DragBar,         TRUE,
    WA_SizeGadget,      TRUE,
    WA_SizeBBottom,     TRUE,
    WA_ReportMouse,     TRUE,
    WA_SimpleRefresh,   TRUE,
    TAG_END,

// Initial group.  No GrpID, no StdID, give it all the weight.
GO_HORIZGRP(0,0,1), TAG_END,

    // Start the first vertical group, which contains buttons 1 to 3.
    GO_VERTGRP(0,0,1), TAG_END,
```

```
            // Create a GadTools box.  The parameters are, in order,
            // GadTools Kind, GrpID, StdID, Weight, Label, Flags.
            GO_GTBOX(BUTTON_KIND, 0, GAD1_ID, 1, (ULONG)&"Gadget One", PLACETEXT_IN),
            GOCT_SetHotKey, '1',  // Assign hotkey '1' to this gadget.
            TAG_END,
            GA_Enabled, TRUE,     // GadTools tag list - make this gadget enabled.
            TAG_END,

            GO_GTBOX(BUTTON_KIND, 0, GAD2_ID, 1, (ULONG)&"Gadget Two", PLACETEXT_IN),
            GOCT_SetHotKey, '2',  // Assign hotkey '2' to this gadget.
            TAG_END,
            GA_Enabled, TRUE,     // GadTools tag list - make this gadget enabled.
            TAG_END,

            GO_GTBOX(BUTTON_KIND, 0, GAD3_ID, 1, (ULONG)&"Gadget Three", PLACETEXT_IN),
            GOCT_SetHotKey, '3',  // Assign hotkey '3' to this gadget.
            TAG_END,
            GA_Enabled, TRUE,     // GadTools tag list - make this gadget enabled.
            TAG_END,

        GO_ENDGRP(),  // End the first vertical group, back into horizontal group.

        // Start the second vertical group, which contains buttons 4 to 6.
        GO_VERTGRP(0,0,1), TAG_END,

            GO_GTBOX(BUTTON_KIND, 0, GAD4_ID, 1, (ULONG)&"Gadget Four", PLACETEXT_IN),
            GOCT_SetHotKey, '4',  // Assign hotkey '4' to this gadget.
            TAG_END,
            GA_Enabled, TRUE,     // GadTools tag list - make this gadget enabled.
            TAG_END,

            GO_GTBOX(BUTTON_KIND, 0, GAD5_ID, 1, (ULONG)&"Gadget Five", PLACETEXT_IN),
            GOCT_SetHotKey, '5',  // Assign hotkey '5' to this gadget.
            TAG_END,
            GA_Enabled, TRUE,     // GadTools tag list - make this gadget enabled.
            TAG_END,

            GO_GTBOX(BUTTON_KIND, 0, GAD6_ID, 1, (ULONG)&"Gadget Six", PLACETEXT_IN),
            GOCT_SetHotKey, '6',  // Assign hotkey '6' to this gadget.
            TAG_END,
            GA_Enabled, TRUE,     // GadTools tag list - make this gadget enabled.
            TAG_END,

        GO_ENDGRP(),  // End second vertical group.

    GO_ENDGRP(),      // End horizontal group.
    GO_ENDOUTLINE()   // End outline array.
    };
```

Basic GadOutline Library Functions

ALLOCATING A GADOUTLINE

Once an outline array has been defined, the function **AllocGadOutline()** is used to parse the array and create an actual **GadOutline** structure based on it. This function is given the address of the outline array, followed by any global outline tags (tags beginning with **GOA_**) which are desired:

```
GadOutline = AllocGadOutline(Outline, GOA_OutlineSize,sizeof(Outline), TAG_END);
```

This returns a pointer to a struct GadOutline, which is a handle on the dynamic definition of this outline. This handle is passed to almost all of the other GadOutline library routines to perform operations on it, and allows the library to track almost every part of the display - including the screen, window, object states and memory.

CREATING A WINDOW

After the GadOutline structure has been allocated, the application still needs to create a window on which the GUI will be displayed using the library function **GO_OpenWindow()**. This function is similar to the Intuition function **OpenWindowTags()**, except that it takes a GadOutline structure to which the window will be attached. Otherwise, all normal window tags may be supplied as per the standard Intuition function; the library will intercept and process any tags it needs to before sending them to Intuition:

```
Window = GO_OpenWindow(GadOutline, WA_PubScreenName,Screen, TAG_END );
```

The function returns a pointer to a **Window** structure, similar to Intuition. Under most circumstances, the program will only be concerned about testing for a **NULL** return, as the library will internally manage the window for the program and attach it to the GadOutline structure.

In addition, any tags specified in the first **GO_WINDOWTAGS()** command in the outline array will be included in the tag list passed to OpenWindowTags(), just as if they had been directly supplied to GO_OpenWindow(). If a tag occurs in both places, the one supplied through GO_OpenWindow() will be used.

The application may call GO_OpenWindow() multiple times. If called again after a window has already been opened, the current window will be closed by the library and a new one opened. This makes it very easy for the application to "move" a window from one screen to another, simply by calling the function with a new screen address.

Alternatively, the current window may be directly closed through **GO_CloseWindow()**:

```
GO_CloseWindow(GadOutline);
```

RECEIVING INTUITION MESSAGES

Similar to GadTools, GadOutline supplies a set of functions on top of Intuition's standard **GetMsg()** and **ReplyMsg()** event handling mechanism, called **GO_GetGOIMsg()** and **GO_ReplyGOIMsg()** respectively. These functions handle hot keys and window resizing and refreshing for the application. Unlike GadTools, though, they return an extended **IntuiMessage** structure, called the **GOIMsg**:

```
struct GOIMsg {
    struct IntuiMessage StdIMsg;    // Standard Intuition message.
    UBYTE KeyPress[8];              // Translated RAWKEY event.
    ULONG UserData;                 // For any use by user.
};  /* There is private data after this structure */
```

The primary addition to the standard IntuiMessage is the inclusion of a **KeyPress** field, which contains a string representing the ASCII conversion of **IDCMP_RAWKEY** events. All normal IntuiMessage fields may be accessed through **StdIMsg**, i.e. StdIMsg.Code.

The two main message handling functions are:

```
GOIMsg = GO_GetGOIMsg(GadOutline);

GO_ReplyGOIMsg(GOIMsg);
```

These functions must be used in place of the standard Intuition functions, and are called in the same message processing loop which traditional Intuition programs use:

```
struct GOIMsg *msg;

while(1) {  // Do forever.

    Wait( (1L<<go->go_MsgPort->mp_SigBit) );

    while (msg = GO_GetGOIMsg(go)) {

        Class = msg->StdIMsg.Class;
        Code = msg->StdIMsg.Code;
        Qualifier = msg->StdIMsg.Qualifier;
        Gadget = msg->StdIMsg.IAddress;

        ....
        // Process event.
        ....

        GO_ReplyGOIMsg(msg);
    }
}
```

MODIFYING A GADGET'S ATTRIBUTES

Two functions are used by the library to dynamically modify an object's attributes, **GO_SetObjAttrs()** and **GO_SetObjGrpAttrs()**. These function operate similarly to the GadTools function **GT_SetGadgetAttrs()** and the BOOPSI function SetGadgetAttrs(), although the state of the object is remembered by the library even when the object doesn't exist. For example, you may set an object's attributes while its window is closed, and upon opening the window these new attributes will take effect.

```
GO_SetObjAttrs(GadOutline,CmdID,Flags, GA_Disabled,TRUE, TAG_END);

GO_SetObjGrpAttrs(GadOutline,CmdID,Flags, GA_Disabled,TRUE, TAG_END);
```

FREEING A GADOUTLINE

The library function **FreeGadOutline()** is used to free everything allocated by a previous call to **AllocGadOutline()**. In addition, any window opened by **GO_OpenWindow()** will be closed and all other resources allocated through the library freed before the GadOutline is deallocated:

```
FreeGadOutline(GadOutline);
```

Function Reference

The following are brief descriptions of the GadOutline functions and macros covered. See the GadOutline autodocs for a complete description and details on each of these calls.

Function	Description
AllocGadOutline()	Create a GadOutline, based on an Outline Array.
FreeGadOutline()	Deallocate a previously allocated GadOutline.
GO_CloseWindow()	Close any window attached to a GadOutline.
GO_GetGOIMsg()	GadOutline compatible version of GetMsg().
GO_OpenWindow()	Open and attach a window to a GadOutline.
GO_ReplyGOIMsg()	GadOutline compatible version of ReplyMsg().
GO_SetObjAttrs()	Set the object attributes of a single command.
GO_SetObjGrpAttrs()	Set object attributes of any commands matching the ID.

Outline Macro	Description
GO_BOOPSIBOX()	Create a BOOPSI gadget.
GO_DRAWBOX()	Create a structured drawing.
GO_EMPTYBOX()	Create a box window nothing in it. (Without an object.)
GO_ENDGRP()	End context of a previously declared group.
GO_ENDOUTLINE()	End parsing of the outline array.
GO_GTBOX()	Create a GadTools gadget.
GO_HORIZGRP()	Begin context of a horizontal group.
GO_OUTLINETAGS()	Set global GadOutline tags. (GOA_ tags.)
GO_VERTGRP()	Begin context of a vertical group.
GO_WINDOWTAGS()	Set global window tags. (WA_ tags.)