foxgui

Simon Fox

COLLABORATORS			
	TITLE : foxgui		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Simon Fox	August 11, 2022	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

foxgui

1.1 FoxGUI Documentation

FoxGUI - The Amiga Graphical User Interface Tools

Version 5.1

1.2 Drag/Drop support in FoxGUI applications

What is drag/drop?

Drag/drop is a way of allowing a user to move data around the user interface in a very visual manner. It's achieved by pointing the mouse at the control/data/image that you want to drag, pressing down and holding down the left mouse button, moving the mouse (with the button still held down) to the point at which you wish to drop the control/data/image and then letting go of the mouse button.

Of course, a user shouldn't really expect to be able to pick up any object on the screen and drag it to just anywhere they like and drag/drop isn't always the most appropriate way of achieving things but it's an increasingly popular option. Here's an example of a situation in which drag/drop might be suitable.

An application was been written to allow customers to order components from a manufacturer. Having reached the screen showing details of the components available (in a list-box for example), the user now has to select which items she wants to order. One way to do this is to have another list-box on the screen showing a list of components ordered. To order a component, the user could drag an item from the available components list-box into the components ordered list-box. If the application is well written it should have alternative ways of doing the same thing, for example double-clicking on an item in the available components list might copy that item into the items ordered list. There might also be an "order component" button on the screen which, when clicked on copies the currently hilighted component in the available components list into the ordered components list.

Not all FoxGUI controls are drag/drop aware. The chart below shows which controls have drag-drop functionality:

Control	Functions supported
Tree Controls	DRAG and DROP
List Boxes	DRAG and DROP
Frames	DRAG and DROP
Windows	DROP only.

How do I use it?

When you create a FoxGUI control you usually pass a "flags" parameter which allows various options to be specified. List boxes, frames and windows have drag/drop flags available which specify whether or not data can be dragged out of or dropped into the control. The flags are summarised below.

Control	Drag flag	Drop flag
Tree Control	TC_DRAG	TC_DROP
List Box	LB_DRAG	LB_DROP
Frame	FM_DRAG	FM_DROP
Window	-	GW_DROP

Although the example application described above allows data to be dragged from one list box into another it is important to realise that data dragged from any control can be dropped into any other drag/drop aware control which was created with the relevant flags set. The originating and receiving controls need not be of the same type. If a control (a list box for example) is created with the LB_DRAG and LB_DROP flags specified then data can be:

* Dragged from that list box to another drag/drop control.
* Dragged from other drag/drop controls into that list box.
* Dragged from that list box into itself!

When you specify drag or drop flags for a control you will also have to create a function which will handle the drag and drop (and possibly other) events. More details about these functions are described in the

MakeListBox
,
MakeFrame
,
OpenGuiWindow
and
MakeTreeControl
function descriptions elsewhere in this documentation but some ↩
general
principles are described below.

Code which is called when a drag begins is typically used to set a pointer to the data being dragged. That pointer will be stored in memory associated with the object that the data is dragged from and can point to absolutely anything you want it to (a struct, an array of items, a FoxGUI control, ... anything). Your drag event should really do nothing other than set that pointer but it can do other things as long as they are FAST! The drag event occurs just as the user starts to drag the data so if the

function takes too long it will hold up the drop function when the user drops the data.

Code which is called when a drop event occurs will typically do something with the data dropped into the control. The drop event function will be passed a pointer which was initialised by the drag event function containing information about what has been dragged. When writing a function to handle a drop event it's worth considering what to do if the data dropped onto the control was dragged from the same control (of course this needn't be considered if the control wasn't created with the relevant flag specified to allow data to be dragged from it). It's worth also considering that if the control also responds to left mouse button clicks then a slight movement of the mouse between pressing and releasing the button will cause up to three events to occur - the left click event, the drag event and the drop event (assuming that all three are enabled for that control). Often, data dragged from the same control should just be ignored but you will need to put code in your drop event to filter out such data.

Drag and drop event functions should either return
GUI_END
or

GUI_CONTINUE
but it should be remembered that in the case of the DRAG event, the return value will be ignored. This makes it impossible to trigger the end of a FoxGUI program at the point a user starts to drag data. Personally I think that would be very bad practise anyway and I can't see why you would want to do it. There are, however, other limitations on what you should do in a drag event.

Drag event limitations

You should not create or destroy FoxGUI controls from within a drag event function (this includes opening and closing windows). There are no such limitations for the DROP event.

1.3 Multi-threading in FoxGUI applications

How to Multi-thread

From release 4.1 onwards, FoxGUI has limited support for multi-threading. What this means is that while your program is performing a lengthy task the user is still free to use other controls in your application and have them respond in the normal way. This is very simple to arrange. Here's an example.

Let's suppose that when the user clicks a certain button in your application, a list of files that the user has selected will be copied from one disk to another. The user may have selected many files and this might take some time. Other buttons in your application might show the contents of a disk or delete a file for example. There's no reason why the user shouldn't be able to do either of these things while the program is copying the original set of files that the user selected.

Here's the function that copies the files:

```
int CopyButtFn(PushButton *copy)
{
   int NumFiles = GetNumFilesToCopy();
   char *Filename = GetFirstFileName();

   while (Filename != NULL)
   {
      CopyFile(Filename);
      Filename = GetNextFileName();
   }

   return GUI_CONTINUE;
}
```

Normally when this function runs, nothing else can happen in your application until this function returns. Any other buttons you click on won't respond until the function has finished.

FoxGUI now has a new function CheckMessages. When you call CheckMessages() FoxGUI will immediately check whether there are any outstanding messages that it should respond to. These could be button clicks, scroll-bar drags, timer events, key presses - anything a user can do that your program should respond to. FoxGUI will process any outstanding events it finds and then the CheckMessages function will return. We can make use of this to allow multi-threading by changing our example above as follows:

```
int CopyButtFn(PushButton *copy)
{
   int NumFiles = GetNumFilesToCopy();
   char *Filename = GetFirstFileName();

   while (Filename != NULL)
   {
      CopyFile(Filename);
      Filename = GetNextFileName();
      CheckMessages();
   }

   return GUI_CONTINUE;
}
```

Now after each file is copied the program will catch up with any other tasks it has been asked to perform before copying the next file.

The pitfalls

Can multi-threading really be that simple? Well, almost. There are one or two pitfalls that are fairly easily avoided but may not be instantly obvious. There are probably more that should be listed here that I haven't thought of yet so I'm sure that over time this list will get longer.

Close Window functions

It may be that your main task (in the example above this would be the CopyButtFn function) opens a window (you might want to have a progress bar in the window which displays the percentage of the task that has completed). In this case the while loop in the example above would also contain calls to the SetProgress function. If the window has a close button then it would be possible for the user to click the close button before the task was complete. The regular calls to CheckMessages() would ensure that in this case the window will get closed before the task has finished. Closing the window will obviously mean destroying the progress bar so after each subsequent file is copied, the SetProgress function will be passed a pointer to a progress bar which has already been destroyed. This is likely to cause severe problems - probably resulting in the computer crashing.

Obviously this problem could be solved by not having a progress bar in the window. Without the progress bar, you may not even need the task to open a window (as in the example above). This is not a very good solution. It's good practice to have some sort of indication of progress when a lengthy task starts otherwise how will the user know that anything's happening?

There are many better solutions to this problem:

* Don't have a close button on the window. The program could automatically close the window once the task is complete. If the user doesn't want the window in the way they can always send it behind other windows.

* Have a close function for the window which checks whether the process is complete and doesn't allow the user to close the window until it is. The process could set a flag at the start and reset it at the end so that the close function could determine whether the process is still running and prevent you from closing the window if it is.

* Have the window's close function set a flag to say that the window has been closed. Modify the main function so that it doesn't call functions (such as SetProgress) which affect controls in the window if the window has already closed. If the process relies on the window being open for reasons other than displaying the progress then this may not be an option.

Repeating the task

Once the main task is underway, regular calls to CheckMessages() mean that the user could start the same task a second time in the same way that they started the first. If for example the task is triggered by clicking on a button, there's nothing to stop the user from clicking the button again and starting a second incarnation of the same task. This may be no problem at all or a complete disaster depending on what the task actually does and exactly how it is coded.

If you do not want the task to be able to start multiple times then there are two simple options to prevent it. The main task could disable whichever button or other control it is that launches the task until the task is complete. Alternatively, it might be appropriate to put the window that contains that control to sleep (by calling the

```
                SleepPointer
               function).  Either method will prevent the user from launching the ↩
                   task
twice simultaneously.

If you do want to be able to launch the task multiple times simultaneously
then this can be handled as long as any windows opened by the task and any
controls that are created in them are declared locally in the function that
launches the task so that each incarnation of the task has it's own copy of
those variables.  This is illustrated below.  In this example, the task is
launced from a button whose click function is shown.

The following would work:

int LaunchTaskButtFn(PushButton *pb)
{
   int i;
   GuiWindow *TaskWindow = OpenGuiWindow(...);
   ProgressBar *ProgBar = NULL;

   if (TaskWindow != NULL)
      ProgBar = MakeProgressBar(TaskWindow, ...);

   // long task...
   for (i = 1; i <= 100; i++)
   {
      // Do Task Stuff
      // ...

      // Update progress bar
      if (ProgBar != NULL)
         SetProgress(ProgBar, i);
   }
   if (ProgBar != NULL)
      Destroy(ProgBar, FALSE);
   if (TaskWindow != NULL)
      CloseGuiWindow(TaskWindow);
   return GUI_CONTINUE;
}

The following would be disastrous:

GuiWindow *TaskWindow;
ProgressBar *ProgBar;

int LaunchTaskButtFn(PushButton *pb)
{
   int i;
   TaskWindow = OpenGuiWindow(...);
   ProgBar = NULL;

   if (TaskWindow != NULL)
      ProgBar = MakeProgressBar(TaskWindow, ...);

   // long task...
   for (i = 1; i <= 100; i++)
   {
```

```
      // Do Task Stuff
      // ...

      // Update progress bar
      if (ProgBar != NULL)
         SetProgress(ProgBar, i);
   }
   if (ProgBar != NULL)
      Destroy(ProgBar, FALSE);
   if (TaskWindow != NULL)
      CloseGuiWindow(TaskWindow);
   return GUI_CONTINUE;
}
```

Limitations of multi-threading in FoxGUI

The major limitation is that the CheckMessages function will not return until all pending messages have been dealt with. This means that if the user triggers a long process and then starts another long process before the first has finished, the first process will not continue until the second has finished even if the second process contains regular calls to CheckMessages. CheckMessages exists to allow the user to perform short tasks while a long task is underway, not to allow multiple long tasks to progress simultaneously.

1.4 Adding your own gadgets to FoxGUI

Hopefully, FoxGUI will one day be able to support bolt-ons so that programmers can build their own gadgets and fully integrate them with FoxGUI. I have already laid the ground work for this but the finished product is still some way off. In the mean time I thought it important that you should be able to devise your own intuition gadgets and use them alongside FoxGUI gadgets in the same window so I have added the capability to do this but there are two limitations:

* FoxGUI gadgets can only be created in FoxGUI windows.
* You cannot manage your own message loop - FoxGUI always does that for you. (To my mind this is an advantage rather than a limitation because it saves you, the programmer, a lot of work but no doubt someone will come up with a genuine reason for wanting to handle their own).

So, how do I do it?

It's easy. There's nothing to stop you from creating your own windows in a FoxGUI program - either right at the start (before you call GuiLoop) or in the event function of a gadget (either a FoxGUI gadget or one of your own) and of course there's nothing to stop you from creating your own gadgets in these windows. The problem is, since FoxGUI handles the message loop for you (in the GuiLoop function) how do you know when an event has occurred that affects one of your gadgets? Well, all you have to do is register your gadget with FoxGUI and then, whenever FoxGUI receives an IntuiMessage (a message from Intuition) which refers to your gadget, a function that you specify will be called. If you create a gadget in a non-FoxGUI window, you

register it by calling the
RegisterGadget
function as follows :

```
RegisterGadget(MyGadget, NULL, MyFunction);
```

where MyGadget is a pointer to the gadget (struct Gadget *) and MyFunction is a pointer to a function of your own which will be called whenever something happens to your gadget. The function you specify should have the following prototype :

```
int MyFunction(struct Gadget *gad, struct IntuiMessage *message);
```

The function will be passed a pointer to the gadget in question and a pointer to the actual IntuiMessage that Intuition sent to FoxGUI. Please note that because this is the original message (not a copy of it) it is important that you don't change it's contents in any way. It is also very important that you ReplyMsg() the message as soon as you have finished with it exactly as you would normally do if you were handling your own message loop (FoxGUI will not do this for you). The function should return either
GUI_CONTINUE or GUI_END
which are described in detail
elsewhere in this manual.

You can register your gadget with FoxGUI either before or after adding it to the window (which you can do using the normal intuition function AddGadget).

If you want to create a gadget in a FoxGUI window you will first need to get a pointer to the Intuition window structure. You do this by calling the function
IntuiWindow
and passing a pointer to the GuiWindow in which
you want to create the gadget e.g.

```
struct Window *MyWindow = IntuiWindow(MyGuiWindow);
```

You now have all the information you need to create your intuition gadget and add it to the FoxGUI window. Be careful to ensure that it doesn't overlap any other gadgets in the window - if the window is resizable and contains auto-sizing FoxGUI gadgets then this will be a problem because at the moment there is no way for your program to find out when a FoxGUI window gets resized.

Now you need to register your new gadget with FoxGUI which you do by calling the
RegisterGadget
function. Unlike registering a gadget in a
non-FoxGUI window, you also need to pass a pointer to the FoxGUI window in which you have placed (or are about to place) the gadget as below:

```
RegisterGadget(MyGadget, MyGuiWindow, MyFunction);
```

When you destroy your gadget (remember that FoxGUI won't do that for you) you must call the
UnRegisterGadget

function so that FoxGUI knows that it
doesn't have to deal with it any more.

1.5 Important Notice.

I make every effort to ensure that if the libraries are used correctly, they will not crash your system or do any other damage. No release is made without first running all of my example FoxGUI programs while enforcer and mungwall are checking that the system is clean but for my own protection, I would like to add that:

The author will not be liable for any damage arising from the failure of this program to perform as described, or any destruction of other programs or data residing on a system attempting to run the program. While the author knows of no damaging errors, the user of this program uses it at his or her own risk.

1.6 Introduction to FoxGUI

What is FoxGUI?

FoxGUI is a shared library of functions that can be called by your C programs (and possibly other languages too if you have the know how) to make a whole host of graphical user interface (GUI) objects very quickly and easily. The objects include screens, windows, buttons, list boxes, edit boxes, drop-down list boxes, menus, file requesters and more. You make use of the objects by calling functions to create them. The functions take all of the necessary parameters for you to customise the objects for your application, including pointers to your own functions which will be called when certain events occur. You then call the GuiLoop function which processes all of the events associated with your objects, calling your functions when necessary. Simple.

For example, the OpenGuiWindow function opens an intuition window. It takes parameters which specify the size and position of the window, the screen that it will open on, whether or not it is dragable, whether it has a close gadget, whether you want a console in the window, a pointer to a function to call when events related to your window occur and many more things besides.

The MakeButton function creates a button! The parameters specify the window which the button will appear in, it's size and position, a user-defined function to call when the button is clicked, whether this function should be called repeatedly if the button is held down etc etc.

Basically, FoxGUI is there so that you as an Amiga programmer can spend more time being inventive and less time writing the run-of-the-mill stuff that lives inside every GUI program. All of the functions are supplied in a shared library.

Why should I use FoxGUI?

Good question. There are other GUI tools available for the Amiga and I guess it really comes down to which one you like. I only have experience of one other (MUI) and I can say without any hesitation that it's main advantage over MUI is speed. It's much faster than MUI but doesn't have quite as many gadgets yet. As for the others? Well, as I say, I haven't used them so I'm not sure. When I started writing this (in 1993) there weren't so many around, in fact I didn't know of any. To help you decide whether FoxGUI is of any use to you, here are some of the things that may distinguish it from other GUIs.

* While making use of functions within the latest versions of the Amiga OS where possible, almost all FoxGUI functions operate on any Amiga so if your code needs to be backwards compatible it can be. You compile one program, it works on any Amiga.

* Unlike MUI, the user of your programs doesn't need to know anything about the GUI. They don't need anything special installed (because you can freely distribute the library with your programs) and don't need to pay to register the GUI. It's completely invisible to them.

* It's fast!

* It's free!

1.7 FoxGUI System Requirements

FoxGUI system requirements

FoxGUI is not very memory hungry so the amount of memory required really depends on the size of your application. It's also quite fast so there's no problem running FoxGUI applications on an unexpanded A500 (I do this all of the time to ensure that any changes I make are backward compatible).

Now that FoxGUI is a shared library (it used to be a link library) there should no-longer be any restriction on what compiler you use in order to create your FoxGUI applications. In fact, there shouldn't even be a restriction on what language you write your programs in. All of the examples in this documentation and on the FoxGUI website are written in C or C++ and will compile with SAS/C 6.58 but should require minimal change to compile them using an alternative C compiler.

See also:
Compatibility

1.8 FoxGUI compatibility

FoxGUI compatibility

I have made every effort to make FoxGUI run on every Amiga ever sold but the earliest Amiga I have access to is a late A500 so there may be Amigas

out there that FoxGUI is incompatible with.

This doesn't mean that I have limited FoxGUI to what is achievable on an unexpanded A500. FoxGUI adapts to the machine that it is run on so that you only have to compile one version of your program and it will work on any Amiga (as long as your code doesn't rely on a later OS)! For example, using FoxGUI, drop down list boxes are available on any Amiga but on an A500+ or above, they can get the focus in the same way that edit boxes can. Once they have the focus you can press a key on the keyboard and if there is an entry in the list that starts with that character it will be selected for you. On an A500, the same program would have the same list box containing the same items but the list box cannot get focus so selection can only be performed by clicking the drop button and selecting using the mouse.

Limitations of each function and how they differ between Amiga models are included in the
functions
section.

1.9 Format of a FoxGUI program

Format of a FoxGUI program

The best way to describe how to write a FoxGUI program is to give a simple example. The code below opens a FoxGUI window on a FoxGUI screen and asks a question which can be answered by selecting a response from a drop-down list box. The program will then respond in an appropriate manner to your response. You can exit the program at any time by pressing the Okay button in the window or by clicking the window's close gadget. The code is heavily commented so that you can hopefully understand what is going on. Assuming you have reasonable knowledge of C, you should find it very simple.

```
#include <clib/exec_protos.h>
#include "FoxGUI.h"
#include "FoxGUIPragma.h"

struct Library *FUIBase;
GuiScreen *GreetingsScreen = NULL;
GuiWindow *GreetingsWindow = NULL;

// CALLBACK is defined in FoxGUI.h

int CALLBACK OkayButtFn(PushButton *pb)
{
   /* We only have one button in our application so we don't need to check which ↩
      one has been clicked.
      We want the Okay button to quit the program so all we have to do is return ↩
         GUI_END and this will
      cause GuiLoop() to return. Our code at the end of the main() function ( ↩
         after the call to GuiLoop)
      will then clear everything up for us. */
   return GUI_END;
```

```
}

int CALLBACK GreetingsWinFn(GuiWindow *win, int event, int x, int y, void *data)
{
   if (event == GW_CLOSE)
   {
      /* We only have one window.  The fact that we are here means that it's close ↩
          gadget has
      been clicked.  We want the close gadget function to actually end the program ↩
          so rather than
      closing the window here, we'll just return GUI_END | GUI_CANCEL (to cause  ↩
         GuiLoop() to exit
      without FoxGUI closing the window for us) and our cleanup code at the end of ↩
          the main()
      function will close this window as well as the screen. */
      return GUI_END | GUI_CANCEL;
   }
   /* We don't want to do anything special for other window events so just return  ↩
      GUI_CONTINUE
      to return control to FoxGUI. */
   return GUI_CONTINUE;
}

BOOL CALLBACK ResponseBoxFn(DDListBox *lb)
{
   /* GetDDListBoxText returns a pointer to the actual buffer used by the drop- ↩
      down list box so we
      mustn't change it directly but it's okay to look at it! */
   char *response = GetDDListBoxText(lb);

   /* Fortunately, all of our entries in the list box begin with different letters ↩
       so instead of
      using strcmp() to work out what has been selected, we can just look at the  ↩
         first letter of the
      text in the drop-down list box. */
   switch (response[0])
   {
      case 'F': // "Fine thanks."
         /* The GuiMessage function displays a modal message to the user.  Modal  ↩
            means that the user
            won't be able to access any gadgets on other windows or screens in  ↩
               this application
            until they have responded to the message which they can do either by  ↩
               clicking on the Okay
            button or by pressing the O or return keys which are hot-keys for the  ↩
               button. */
         GuiMessage(GreetingsScreen, "I'm glad to hear it.", "FoxGUI", 2, 1,  ↩
            GM_OKAY);
         break;
      case 'T': // "Terrible!"
         GuiMessage(GreetingsScreen, "I'm sorry to hear that!", "FoxGUI", 2, 1,  ↩
            GM_OKAY);
         break;
      case 'N': // "None of your business!"
         GuiMessage(GreetingsScreen, "Well there's no need to be rude!", "FoxGUI", ↩
             2, 1, GM_OKAY);
   }
```

```
   return TRUE;
}

/* This function closes all of the things that we opened in main() to free up the ↩
   memory and other
   system resources because our program is about to exit. */

static int CloseDown(int retval)
{
   if (GreetingsWindow)
   {
      /* ResponseBox is declared in main() so we don't have a pointer to it here ↩
         and so we can't
         use the Destroy() function.  However, we can destroy it by calling ↩
            DestroyM() with a
         pointer to our window and a type of DDListBoxTypeID.  That will destroy ↩
            any drop-down list
         boxes in that window and is actually much more convenient because it ↩
            saves us from checking
         whether the drop-down list box was created successfully in the first ↩
            place (this function
         will be called if the MakeDDListBox() function in main() fails).  Exactly ↩
             the same is true
         of our button (OkayButton) so we use the DestroyM() function again ↩
            instead of the Destroy()
         function. */

      DestroyM(DDListBoxTypeID, GreetingsWindow, FALSE);
      DestroyM(ButtonTypeID, GreetingsWindow, FALSE);

      /* Close the window we created in the main() function. */
      Destroy(GreetingsWindow, TRUE);
   }

   /* Close our screen (if it opened successfully). */

   if (GreetingsScreen)
      Destroy(GreetingsScreen, TRUE);

   /* Free up all resources used by the Gui */

   CloseLibrary(FUIBase);

   return retval;
}

int main(void)
{
   DDListBox *ResponseBox = NULL;
   PushButton *OkayButton = NULL;

   /* Initialise FoxGUI.  It is essential that this is done before any other calls ↩
       are made to FoxGUI
      functions. */

   FUIBase = OpenLibrary("FoxGUI.library", 0);
   if (!FUIBase)
```

```
    return 1;

/* Open a GuiScreen to run our application on.  We'll just give it two ↩
    bitplanes which allows
    four colours - plenty for such a simple program. */

if ((GreetingsScreen = OpenGuiScreen(2, 2, 1, "Welcome to FoxGUI", NULL, 0,  ↩
    NULL, 0, 0, NULL, NULL)) == NULL)
    return CloseDown(1);

/* Open the GuiWindow to create our controls in.  By supplying the GW_CLOSE  ↩
    parameter we ensure that
    the window has a close gadget and by supplying a pointer to our  ↩
       GreetingsWinFn() (defined
    above) we ensure that if the user clicks the close gadget, FoxGUI won't just ↩
        close the window
    for us but will call our function instead and do what ever our function  ↩
       tells it to do! */

if ((GreetingsWindow = OpenGuiWindow(GreetingsScreen, 100, 40, 400, 100, 2, 1,  ↩
    "Greetings!",
            GW_DRAG | GW_CLOSE, GreetingsWinFn, NULL)) == NULL)
    return CloseDown(1);

/* Make the drop-down list box that the user will use to select a response to  ↩
    the question.  Notice
    that I've specified the question itself (How are you today?) as the PreText  ↩
       for the drop-
    down list.  There are many other ways I could have done this (using an  ↩
       output box or using a
    console and writing text directly to the window or even by bypassing FoxGUI  ↩
       altogether and
    allocating my own IntuiText and writing it to the window's rastport) but  ↩
       since I wanted the text
    to be to the left of the list box, this was by far the easiest method.  We' ↩
       ll add the possible
    responses into the list box in a moment. */

if ((ResponseBox = MakeDDListBox(GreetingsWindow, 160, 30, 190, 22, 3, 0,
            ResponseBoxFn, THREED, NULL)) == NULL)
    return CloseDown(1);
SetPreText(ResponseBox, "How are you today?");

/* Make the Okay button which the user can click to quit the application.   ↩
    Notice that we pass a
    pointer to our OkayButtFn() defined above which will get called when the  ↩
       user clicks the okay
    button.  Notice also that I've set the hot-key to 'O' and underlined the O  ↩
       of Okay so that the
    user knows what the hot-key is.  I've also passed the BN_OKAY flag to tell  ↩
       FoxGUI to allow the
    return key as an extra hot-key for this button. */

if ((OkayButton = MakeButton(GreetingsWindow, "_Okay", 170, 80, 60, 14, 'O',  ↩
    NULL, OkayButtFn,
            BN_CLEAR | BN_STD | BN_OKAY, NULL)) == NULL)
    return CloseDown(1);
```

```
   /* Add the three possible responses to our drop-down list box. */

   AddToDDListBox(ResponseBox, "Fine thanks");
   AddToDDListBox(ResponseBox, "Terrible!");
   AddToDDListBox(ResponseBox, "None of your business!");

   GuiLoop();

   /* It is important to free up any memory we have used by destroying all of the  ↩
      FoxGUI gadgets we
      have created.  All of this is done in the CloseDown() function. */

   return CloseDown(0);
}
```

1.10 Suggestions?

Suggestions?

If you have any suggestions for improvements you would like to see in the next release of FoxGUI, or if you have any other need to contact me I can be reached at simon@foxysoft.co.uk. Please don't expect immediate replies as I'm usually too busy improving FoxGUI to actually check my mail! I will get back to you as soon as I can.

Also, keep an eye on the web page. Details of forthcoming releases and improvements that are planned are listed there:

http://www.foxysoft.co.uk/

1.11 FoxGUI Bugs

Bugs in the current release of FoxGUI

The following problems are known to exist in the current release. If you find any others please Email me at: simon@foxysoft.co.uk

* The GW_CONSOLE flag to the function OpenGuiWindow currently doesn't work. I have had problems implementing consoles in the shared library version of FoxGUI but I hope to have this cleared up soon.

* If you create a frame (using the function MakeFrame) and you specify the FM_DRAG option but do not set your own custom drag pointer using the function SetFrameDragPointer() then dragging may cause a crash. You can overcome this by putting the following line of code immediately after your call to MakeFrame:

  ```
  myFrame->DragPointer = NULL; // Fix the bug in MakeFrame
  ```

  This bug will be fixed in the next release.

Keep your eye on the Web page for info about future releases
(http://www.foxysoft.co.uk/).

1.12 FoxGUI Wish list

What's next for FoxGUI

There is still much that can be done to improve FoxGUI. There are plenty of things that I intend to implement as soon as I get time and just so that you know what's coming soon, I've outlined them below. They are in approximate order of importance but subject to change.

* Reinstate the console functions.
* Improvements to frames
* Pop-up menus.

1.13 FoxGUI functions

FoxGUI Functions

Since there are rather a lot of FoxGUI functions, I have broken them down into sections divided by the object which they affect. Each section contains a description of the object type and full descriptions of each function available for objects of that type :-

Screens

Windows

Menus

Tab controls

Frames

Buttons

Boolean gadgets

Edit boxes

List boxes

Tree Controls

Drop-down list boxes

Output boxes

Timers

Progress Bars

Images

Miscelaneous

1.14 FoxGUI Image functions

FoxGUI has a number of functions for manipulating ILBM images. Any ILBM image can be loaded, shown in a FoxGUI window, attached to a button or frame etc.

All of the image functions require IFFparse.library. Two versions of IFFparse.library are available - one will have been on your workbench disk if you bought a V39 Amiga (e.g. an Amiga 1200) which only works on Amigas with V39 and above. The other works on all Amigas all the way back to an A500 with workbench 1.3 but was only supplied with Amigas from V37 onwards so if you have an older Amiga then you will need to get hold of that version. Unfortunately I don't have a license to distribute it so I can't supply it with FoxGUI but you may find that you already have it supplied with some other piece of software.

If you make use of the FoxGUI image functions in your code and someone attempts to run the code on an Amiga without IFFparse library, the program won't fail or crash or do anything nasty - the user just won't see your images.

The following image functions are currently available :-

LoadBitMap

ShowBitMap

HideBitMap

FreeGuiBitMap

ScaleBitMap

RedrawBitMap

AttachBitMapToControl

ScreenColoursFromILBM

1.15 LoadBitMap function

Function prototype:

```
GuiBitMap *LoadBitMap(char *fname);
```

Description:

Loads an ILBM image with the specified path and filename and returns a GuiBitMap structure which can be passed as a parameter to other FoxGUI image functions.

Parameters:

fname: The full or relative path and filename of the ILBM image to load.

Returns:

If successful, a pointer to a GuiBitMap structure containing the image loaded, otherwise NULL. This function will fail if the IFFparse library could not be opened.

Known bugs:

None.

See also:

ShowBitMap

HideBitMap

FreeGuiBitMap

ScaleBitMap

RedrawBitMap

AttachBitMapToControl

ScreenColoursFromILBM

1.16 ShowBitMap function

Function prototype:

```
BitMapInstance *ShowBitMap(GuiBitMap *bm, GuiWindow *w, unsigned short
   x, unsigned short y, short flags);
```

Description:

Displays a previously loaded image in a FoxGUI window. Note that the image won't look as it was intended to unless your screen is deep enough

```
   to show the correct number of colours for the image and the screen's
   pallette matches the images pallette (see
                ScreenColoursFromILBM
                  and
                OpenGuiScreen
               ).

Parameters:

      bm: A pointer to a GuiBitMap structure as returned by the

                LoadBitMap
                function.
       w: A pointer to the FoxGUI window in which to display the image.
    x, y: The coordinates (in pixels) within the window for the top left
          hand corner of the image.
   flags: The only flag currently available for this function is
          BM_OVERLAY.  If this is specified, any pixels within the image
          which don't contain any data (i.e. are transparent) won't be
          drawn.  This allows an image to be shown over the top of another
          and the bottom image to be seen "through" the top one where it is
          transparent.  If this flag is not specified then every pixel in
          the image will be drawn so anything currently drawn in the window
          will be overwritten if it falls within the bounds of the image.

Returns:

   If successful, a pointer to a BitMapInstance is returned.  Keep a copy
   of this - you will need it to free the memory used by the image or to
   hide or refresh it.

Known bugs:

   None.

See also:


                LoadBitMap

                HideBitMap

                FreeGuiBitMap

                ScaleBitMap

                RedrawBitMap

                AttachBitMapToControl

                ScreenColoursFromILBM
```

1.17 HideBitMap function

```
                Function prototype:

   BOOL HideBitMap(BitMapInstance *bmi);

Description:

   Hides an image previously displayed in a window with the
                ShowBitMap
                  function and free's the memory used by the BitMapInstance.  You ↩
                      should
   never refer to a BitMapInstance when it has been hidden as that memory
   will have been free'd - if you show the image again (using ShowBitMap)
   you will be given a new BitMapInstance.  If the image was shown on top
   of other images as an overlay, you will need to refresh the other images
   using
                RedrawBitMap
               .

Parameters:

   bmi: A pointer to the BitMapInstance returned by ShowBitMap when the
        image was drawn.

Returns:

   TRUE if the image was successfully hidden, FALSE otherwise.

Known bugs:

   None.

See also:


                LoadBitMap

                ShowBitMap

                FreeGuiBitMap

                ScaleBitMap

                RedrawBitMap

                AttachBitMapToControl

                ScreenColoursFromILBM
```

1.18 FreeGuiBitMap function

```
                Function prototype:

   BOOL FreeGuiBitMap(GuiBitMap *bm);
```

Description:

This function free's the memory used by an image that was loaded with the LoadBitMap function. You should not free a GuiBitMap while it is displayed (other than attached to a control). Only call this function when you have completely finished with the image.

Parameters:

bm: A pointer to the GuiBitMap as returned by the LoadBitMap function.

Returns:

TRUE if the image was successfully free'd. FALSE otherwise.

Known bugs:

None.

See also:

LoadBitMap

ShowBitMap

HideBitMap

ScaleBitMap

RedrawBitMap

AttachBitMapToControl

ScreenColoursFromILBM

1.19 ScaleBitMap function

Function prototype:

```
GuiBitMap *ScaleBitMap(GuiBitMap *source, unsigned short destwidth,
   unsigned short destheight);
```

Description:

Scales an image loaded with LoadBitMap to the size specified. Note that this function requires graphics library V36 or above and will fail when used on V35 or below.

```
Parameters:

       source: A pointer to a GuiBitMap to scale.
    destwidth: The target width in pixels of the scaled image.
   destheight: The target height in pixels of the scaled image.

Returns:

   If successful, a pointer to a new GuiBitMap is returned which is a
   scaled version of the one supplied.  Note that if the original image is
   not displayed or attached to a control then the original need not be
   kept after calling this function (i.e.  you can free it with a call to

                FreeGuiBitMap
               ).  NULL is returned if this function fails.

Known bugs:

   None.

See also:

                LoadBitMap

                ShowBitMap

                HideBitMap

                FreeGuiBitMap

                RedrawBitMap

                AttachBitMapToControl

                ScreenColoursFromILBM
```

1.20 RedrawBitMap function

```
               Function prototype:

   BOOL RedrawBitMap(BitMapInstance *bmi);

Description:

   Redraws the BitMapInstance supplied.  If the image was originally drawn
   as an overlay then the refresh will also draw it as an overlay.

Parameters:

   A pointer to a BitMapInstance as returned by
                ShowBitMap
               .
```

```
Returns:

   TRUE if successful, FALSE otherwise.

Known bugs:

   None.

See also:

                LoadBitMap

                ShowBitMap

                HideBitMap

                FreeGuiBitMap

                ScaleBitMap

                AttachBitMapToControl

                ScreenColoursFromILBM
```

1.21 AttachBitMapToControl function

```
               Function prototype:

   BOOL AttachBitMapToControl(GuiBitMap *bm, void *control, short left,
      short top, short width, short height, int flags);

Description:

   Attaches an image (as loaded with
                LoadBitMap
               ) or part of an image to a
   FoxGUI button or frame.  Many images can be attached to the same control
   with multiple calls to this function.

Parameters:

        bm: A pointer to the GuiBitMap image to attach to the control.
            Note that a copy is made of the required portion of the image
            so there is no need to maintain the original after calling this
            function if it is not needed elsewhere.
   control: A pointer to a FoxGUI frame or button to which to attach the
            image.
   left, top, width, height:
            These specify the portion of the image to attach to the
            control.  left and top specify the coordinates within the image
            of the top left corner of the portion of the image to attach to
            the control.  width and height specify the width and height of
            the portion that you want to attach (in pixels).  width or
```

height can be set to -1 which means the rest of the width/height of the image. For example, to attach the whole image to the control, specify left, top, width and height as 0, 0, -1 and -1 respectively. To attach all of the image except the top 5 rows and the left 3 columns, specify 3, 5, -1, -1.

flags: The following flags are available :- BM_OVERLAY, BM_SCALE, BM_CLIP, BM_SMART and BM_STUPID. BM_OVERLAY causes the image to be over-laid onto the control. This means that any pixels in the image which are transparent won't get drawn, allowing several images to be attached to the same control and to be transparent. BM_SCALE causes the image to be scaled to fit the control but only works on Amigas with graphics library version 36 or above. On Amigas with graphics library version 35 or below, BM_SCALE is ignored and BM_CLIP is used instead. BM_CLIP is the opposite of BM_SCALE and is the default if neither is specified. BM_CLIP causes the image to be clipped to fit the control. BM_SMART can be used when attaching images to controls which have the S_AUTO_SIZE flag set. It causes an extra copy of the image to be kept so that if a window containing the control is resized (causing the control to resize), the image attached to the control will be re-clipped or re-scaled (depending upon whether BM_CLIP or BM_SCALE was specified) from the original. BM_STUPID is the opposite of BM_SMART and is the default if neither is specified. When a BM_STUPID control is resized, the image will still be re-clipped or re-scaled but from the image currently shown on the control - not from the original. The result is that if a control containing a clipped image gets larger, no extra will be shown or if the image is scaled, repeated re-scaling of the image will cause quite rapid loss of definition because each rescale will be done from the previously scaled version. If BM_SMART is specified when attaching an image to a control which doesn't have the S_AUTO_SIZE flag set (and hence will never change size) it is ignored and BM_STUPID is used instead. The advantage of BM_SMART is obvious. The advantage of BM_STUPID is that it requires less memory.

Returns:

TRUE for success, FALSE for failure.

Known bugs:

None.

See also:

LoadBitMap

ShowBitMap

HideBitMap

FreeGuiBitMap

ScaleBitMap

RedrawBitMap

ScreenColoursFromILBM

1.22 ScreenColoursFromILBM function

Function prototype:

```
BOOL ScreenColoursFromILBM(GuiScreen *sc, char *fname);
```

Description:

Modifies the pallette of an open FoxGUI screen to match the pallette of the ILBM image whose filename is specified. Useful for applications such as slideshow viewers where it will be necessary to constantly review the pallette to match each slide. Note that this function will affect everything displayed on the screen and will spoil the 3D effect of any controls on the screen unless the shine and shadow pens in the image happen to match those used for the screen. This function attempts to set the shine and shadow pens to sensible colours by calling

SetGuiPens
but it's not currently very clever so you may have to call the function again yourself afterwards. Also note that SetGuiPens doesn't redraw any existing controls so if you're going to call ScreenColoursFromILBM it's best to do it before drawing any controls.

Parameters:

sc: A pointer to an open FoxGUI screen.
fname: The filename of an ILBM image whose pallette is to be copied.

Returns:

TRUE for success, FALSE for failure. Note that this function will fail if the screen isn't deep enough for the number of colours in the images pallette.

Known bugs:

None.

See also:

LoadBitMap

1.23 FoxGUI Progress Bar functions

Progress bars are indicators to display how far through a process you are.

Usually, when an application is busy it will change the pointer for it's window to a stop-clock so that the user knows it's busy. If it's busy doing something that will take a long time, it's polite to show the user how much progress is being made so that he knows the machine hasn't crashed and whether he's got time to go and make a cup of tea (or even have a nights sleep) before the application is ready for input again. This is what progress bars are for. Typically they consist of a long, thin rectangular frame which gradually gets filled in a different colour to show progress. When the frame is full, the processing has finished and the user can gain some idea of how long he has to wait by how full the progress bar is.

The following progress bar functions are currently available :-

MakeProgressBar

SetProgress

SetProgressMax

See Also:

Destroy

Hide

Show

1.24 MakeProgressBar function

Function prototype:

```
ProgressBar *MakeProgressBar(void *Parent, int left, int top, int width,
   int height, short fillcol, short flags, void *extension);
```

Description:

Make a new progress bar in the specified FoxGUI window or frame. The progress indicator will initially be set to zero.

Parameters:

```
 Parent: A pointer to a FoxGUI window or frame in which to make the
         progress bar.
   left: The offset of the left edge of the progress bar (in pixels)
         from the left edge of the window/frame.
    top: The offset of the top edge of the progress bar (in pixels)
         from the top edge of the window/frame.
  width: The width of the progress bar in pixels.
 height: The height of the progress bar in pixels.
```

```
   fillcol: The colour in which to fill the progress bar to indicate
            progress.
     flags: The currently available flags are :-
            PB_RAISED, PB_INSET : These specify whether the progress bar
            appears rased from or inset into the screen.  The default is
            raised if neither is specified.
            PB_CAPTION_CENTRE, PB_CAPTION_TOP_LEFT, PB_CAPTION_BOTTOM_LEFT,
            PB_CAPTION_TOP_RIGHT, PB_CAPTION_BOTTOM_RIGHT : These specify
            the position of the caption which can go above or below and
            left or right of the progress bar or can even go right in the
            middle of the bar.  If you want the text in the middle, you
            should make sure that tcol and fillcol are not the same colour.
            If none of these is specified then the default is top right.
            PB_FILL_LR, PB_FILL_BT : These specify whether the progress bar
            should fill from the left towards the right or from the bottom
            upwards.  The default is left to right.

                 S_AUTO_SIZE
                 extension: This is for future expansion.  Always set this to NULL ↩
                    .

Returns:

   If successful, a pointer to a new progress bar is returned.  If not,
   NULL is returned.

Known bugs:

   None.

See also:


                 SetPreText

                 SetPostText

                 SetProgress

                 SetProgressMax

                 Destroy
```

1.25 SetProgress function

```
               Function prototype:

   void SetProgress(ProgressBar *pb, int progress);

Description:

   Sets the current amount of progress for the specified progress bar.

Parameters:
```

```
        pb: A pointer to a progress bar to update.
  progress: The amount of progress that has been made.  This should be
            between zero and the maximum progress amount for the progress
            bar.  The maximum is generally 100 so that progress is shown as
            a percentage but it can be set to anything you like by calling

                 SetProgressMax
                .

Known bugs:

   None.

See also:


                 MakeProgressBar

                 SetProgressMax
```

1.26 SetProgressMax function

```
                Function prototype:

   void SetProgressMax(ProgressBar *pb, int progressmax);

Description:

   Set the maximum for a progress bar.  By default, progress bars show
   progress as a percentage so you can call
                 SetProgress
                 with any number
   between 0 and 100.  This is not always useful, however.  For example,
   let's take the case of an internet browser which downloads messages from
   news groups and has to put them in your local database.  It could use a
   progress bar to show progress in adding them to the database.  If the
   progress bar went from 0 to 100, the programmer would need to work out,
   after processing each message, what percentage of messages had been
   processed and then set the progress bar accordingly.  It would be much
   easier, however, to use this function to set the maximum to the number
   of messages downloaded and then just add one to the progress indicator
   after processing each message.

Parameters:

            pb: A pointer to the progress bar to modify.
   progressmax: The maximum for the progress bar.

Known bugs:

   None.

See also:
```

MakeProgressBar

SetProgress

1.27 FoxGUI Frame functions

Frames are rather like buttons in that they look like buttons and that if you click on them, they perform a function which you define. Like buttons they can also have images attached to them (drawn on them) - see

AttachBitMapToControl

and they can be hidden or shown at will. Unlike buttons, however, frames can also respond to right mouse button clicks.

Frames are also rather like windows in that controls can be created within them. When you create a control, one of the parameters to the function is a pointer to a window in which to create the control. Instead of passing a pointer to a window you can pass a pointer to a frame if you prefer. The control will be created inside the frame, offset from the top left hand corner by the "left" and "top" parameters you specify. The advantage of creating controls in frames is that a group of associated controls can be created in the same frame which gives a border around the group and shows visually that they are associated with each other. (Of course, frames do not have to have a border if you don't want one). Another advantage is that if you then want to hide a whole group of controls in the same frame, you can just hide the frame - all of the controls within it will become hidden too!

The following frame functions are available :-

MakeFrame

SetFrameDragPointer

See also:

Destroy

DisableControl

EnableControl

Hide

Show

SetPreText

SetPostText

1.28 MakeFrame function

```
                Function prototype:

   Frame *MakeFrame(void *Parent, char *name, int left, int top, int width,
      int height, struct Border *cb, int (*callfn)
      (Frame*, short, short, short, void**), short flags, void *extension);

Description:

   Make a new FoxGUI frame in the specified FoxGUI window or frame.

Parameters:

   Parent, left, top, width, height, cb:
      These parameters are identical to the similarly named parameters of

                 MakeButton
                .

     name: As with MakeButton, this should be a pointer to a NULL
           terminated text string to use as a caption for the frame.  If
           you do not want a caption you should use "" or NULL.  Unlike
           buttons however, frames cannot have hot-keys and so you cannot
           use the "_" character to indicate a character in the caption to
           underline.  You can of course use the "_" character in your
           frame caption - it just won't act the way it does for buttons.
           If the caption for the frame is too long to fit in the frame it
           will be truncated to fit - the caption will never extend beyond
           the frame border.  If the frame gets resized (due to a window
           being resized) then more of the caption may become visible -
           i.e.  FoxGUI remembers the whole caption not just the visible
           bit.
   callfn: A pointer to a function to call when the user clicks on your
           frame with the left or right mouse button or data is dragged
           into or out of this frame (depending on which flags are
           specified).  The function should have the following prototype :-

           int CALLBACK MyFrameFn(Frame *Fm, short Event, short x,
              short y, void **DragData);

           Fm will be a pointer to the frame which was clicked/dragged by
           the user.  Event will have one of the values FM_LBUT, FM_RBUT,
           FM_DRAG or FM_DROP depending on whether the left or right mouse
           button was clicked or data was dragged out of or dropped into
           the frame respectively.  x and y will contain the coordinates of
           the exact position of the mouse event (left click, right click,
           drag or drop) relative to the top, left hand corner of the
           frame.  DragData is NULL unless the Event is FM_DRAG or FM_DROP.
           If the user is dragging this frame then you will probably want
           to store a pointer to the data that's being dragged somewhere
           where it can easily be found by the control that the data is
           eventually dropped in.  The data can be anything you want.
```

DragData is a pointer to a pointer that you can modify to point to whatever piece of data it is that the user is dragging. Let's take a simple example - you have an application with two frames and you want to be able to drag the name of either frame to the other. Your event function for one of your frames might look like this:

```
int FirstFrameFn(Frame *Fm, short Event, short x, short y, void ** ↩
   DragData)
{
   if (Event == FM_DRAG)
   {
      /* The user is dragging the name of this frame to another
         control (or maybe to itself!).  Set DragData to point
         to a constant text string containing the name of this
         frame (Frame1). */
      *DragData = "Frame1";
   }
   else if (Event == FM_DROP)
   {
      /* The user has dropped something in this frame.  Let's
         tell the user what has been dropped and where! */
      char Message[100];
      sprintf(message, "%s was dropped in Frame1 at (%d, %d)", * ↩
         DragData, x, y);
      GuiMessage(MyScreen, Message, "Drop!", 1, 2, 6, 5, GM_OKAY);
   }
return GUI_CONTINUE;
}
```

As you can see from the function above, it is actually *DragData that you set and use - DragData is a pointer to a pointer that you can modify. See the notes in the
Drag/Drop functionality
section for more information about drag and drop event functions.
Your function should return either
GUI_END
or
GUI_CONTINUE
.

flags: The following flags are available: FM_LBUT, FM_RBUT,
S_AUTO_SIZE
,
FM_CLEAR, FM_DRAG, FM_BORDERLESS, FM_DRAGOUTLINE and FM_DROP. FM_LBUT causes the frame to respond to left button clicks and FM_RBUT causes it to respond to right button clicks. FM_CLEAR is the same as the BN_CLEAR flag for buttons (see
MakeButton
).
The FM_DRAG flag indicates that the user will be able to drag data from this frame into other drag/drop aware controls and the FM_DROP flag indicates that the user will be able to drag data into this control from other drag/drop aware controls. The FM_BORDERLESS flag causes the new frame to have no border. The FM_DRAGOUTLINE flag will cause the outline of the frame to be moved with the pointer when dragging the frame (this flag will

```
           be ignored if the FM_DRAG flag has not been specified).
extension: This is for future expansion.  Always set this to NULL.

Returns:

   If successful, a pointer to a new FoxGUI frame.

Known bugs:

   None.

See also:


                SetFrameDragPointer

                Drag/Drop functionality

                Destroy

                Hide

                AttachBitMapToControl
```

1.29 SetFrameDragPointer function

```
               Function prototype:

   void SetFrameDragPointer(Frame *Fptr, unsigned short *DragPointer, int width,
      int height, int xoffset, int yoffset)

Description:

   Specify a custom mouse pointer to be used when data is being dragged
   from this frame.

Parameters:

          Fptr: A pointer to the frame whose drag pointer is to be set.
   DragPointer: A pointer to an array of numbers making up a standard
                Intuition sprite data structure.  This must be stored in
                chip memory since it is to be used as a mouse pointer.
         width: The width in pixels of the pointer provided.  The maximum
                width of an Amiga mouse pointer is 16 pixels.
        height: The height in pixels of the pointer provided.  There is no
                maximum height.
       xoffset:
       yoffset: These two numbers specify the offset of the pointers
                hot-spot from the top left corner of the sprite.  They are
                typically zero or negative.

Known bugs:

   None.
```

See also:

MakeFrame

1.30 FoxGUI Timer functions

Timer controls allow your program to perform tasks at regular intervals (every second or every minute). You do this by creating a timer using the

MakeTimer

function which you supply with a pointer to a function that you want to be called and a flag specifying whether you want it called every second or every minute. The timer will not be active (and will not call your function) until you start it with the

StartTimer

function and will not stop until you call either the

StopTimer

or

PauseTimer

function. If your timer function is to be called every second then it's vital to make sure that the code in your function returns as fast as possible so as not to cause your Amiga to grind to a halt under the strain and if your function takes over a second to run then it will not be called again for the second that was missed during your functions execution. If your timer is only going to trigger your function once a minute then these issues won't apply. Always destroy a timer (using

Destroy

) when you've finished with it so that it doesn't continue to use up valuable processing time.

The following Timer functions are currently available :-

AddTime

MakeTimer

PauseTimer

SetTime

StartTimer

StopTimer

UnpauseTimer

See also:

Destroy

1.31 AddTime function

Function prototype:

```
   void AddTime(Timer *t, long secs);
```

Description:

Adds a specified number of seconds to the running time of the specified timer. For example, if timer T had been runnning for 10 seconds when you called AddTime(T, 50) then it would now behave exactly as if it had been running for 1 minute.

Parameters:

```
      t: The timer whose time is to be modified.
   secs: The number of seconds to add to timer t.  Note that this can be a
         negative number if you wish to subtract time.
```

Known bugs:

None.

See also:

MakeTimer

PauseTimer

SetTime

StartTimer

StopTimer

UnpauseTimer

1.32 MakeTimer function

Function prototype:

```
   Timer *MakeTimer(short flags, int (*CallFn) (Timer *, long),
      void *extension);
```

Description:

Create a new timer control.

```
Parameters:

    flags: Two flags are available: TM_SECOND and TM_MINUTE.  Only one of
           these should be specified.  If you specify TM_SECOND, your timer
           function will be called once a second.  If you specify TM_MINUTE,
           your timer function will be called once a minute.
   CallFn: A pointer to a function to call once a second or once a minute
           while the timer is running.  At any time when the timer is
           stopped or paused, the function won't be called.  The function
           should have the following prototype:

           int CALLBACK MyTimerFunction(Timer *WhichTimer, long
              TimeInSeconds)

           Your function will be passed a pointer to the timer that
           triggered it (you can have more than one timer calling the same
           function if you like) and the time (in seconds) since the timer
           was started (excluding any time during which the timer was
           paused).  Your function should return either
                 GUI_END
                 or

                 GUI_CONTINUE
                .
extension: This is reserved for future expansion and should be set to NULL.

Returns:

  If successful, a pointer to the new timer control.  NULL otherwise.

Known bugs:

   None.

See also:


                 AddTime

                 Destroy

                 PauseTimer

                 SetTime

                 StartTimer

                 StopTimer

                 UnpauseTimer
```

1.33 PauseTimer function

```
                Function prototype:

   void PauseTimer(Timer *t);

Description:

   Pauses the specified timer (assuming it is currently running).  When a
   timer is paused, your timer function will not get called at regular
   intervals as it would while the timer is running.  However, unlike
   stopping a timer with the
                StopTimer
                function, the number of seconds
   that the timer has been running is retained and when the timer is
   unpaused using the
                UnpauseTimer
                function it will continue counting
   the seconds from where it left off.

Parameters:

   t: A pointer to the timer to pause.

Known bugs:

   None.

See also:


                AddTime

                SetTime

                StartTimer

                StopTimer

                UnpauseTimer
```

1.34 SetTime function

```
                Function prototype:

   void SetTime(Timer *t, long secs);

Description:

   This function is used to set the elapsed time (the time that a timer
   has been running).

Parameters:

      t: The timer control whose elapsed time you wish to set.
```

```
   secs: The time (in seconds) to which to set the elapsed time.

Known bugs:

   Currently, this function cannot be used to set the time to 0.

See also:

                AddTime

                PauseTimer

                SetTime

                StartTimer

                StopTimer

                UnpauseTimer
```

1.35 StartTimer function

```
               Function prototype:

   void StartTimer(Timer *t);

Description:

   Starts the specified timer.  If the timer is paused and you want to
   restart it without resetting the elapsed time to zero then use

                UnpauseTimer
                instead of this function.  StartTimer always resets the
   elapsed time to zero.

Parameters:

   t: The timer to start.

Known bugs:

   None.

See also:

                PauseTimer

                SetTime

                StopTimer

                UnpauseTimer
```

1.36 StopTimer function

```
                Function prototype:

   void StopTimer(Timer *t);

Description:

   Stops the specified timer (assuming that it is running).  If you wish to
   stop a timer and restart it later with the elapsed time preserved then
   you should use
                PauseTimer
                instead of this function to stop the timer.
   When a timer is stopped or paused, the timer function isn't called.

Parameters:

   t: The timer to stop.

Known bugs:

   None.

See also:


                PauseTimer

                StartTimer
```

1.37 UnpauseTimer function

```
                Function prototype:

   void UnpauseTimer(Timer *t);

Description:

   Restart a paused timer, preserving the elapsed time as it was when the
   timer was paused.

Parameters:

   t: The paused timer which you wish to "unpause".

Known bugs:

   None.
```

See also:

PauseTimer

StartTimer

StopTimer

1.38 FoxGUI Screen functions

All Amiga programs run on what is called a "screen". Initially a screen may have nothing on it other than a title bar across the top. Generally speaking, an application will open all of it's windows on one screen but this is not compulsory. An application may, if appropriate, open more than one screen or may choose not to open one at all but run on a public screen such as the workbench screen. If you wish to have control over the resolution of your applications display and the number of available colours then the simplest way is to open your own screen because that is where these attributes are defined.

FoxGUI screens are standard Intuition (Amiga) screens. This means that the user can control FoxGUI screens in the same way as any other applications screens. Left Amiga-m can be used to flick between screens and left Amiga-n can be used to bring the workbench screen to the front.

The following screen functions are currently available :-

ClonePublicScreen

GetModeName

GetModeSize

GetNextAvailableDisplayMode

OpenGuiScreen

ShowDisplayList

See also:

Destroy

1.39 ClonePublicScreen function

Function prototype:

```
   GuiScreen *ClonePublicScreen(int mindepth, UBYTE *pub_screen_name, char
      *sScreenTitle, int (* __far __stdargs LastWinFn)(GuiScreen *), int
      flags, char *new_pub_name, int OverscanType, void *extension)

Description:

   Open a new screen based on a public screen.  Requires V36.

Parameters:

        mindepth: The minimum number of planes for the screen.  This
                  determines the number of colours that can be shown on the
                  screen (i.e.  1 plane gives 2 colours, 2 planes give four
                  colours, 3 planes give 8 colours etc).  The screen being
                  opened will have the depth specified or the depth of the
                  public screen being cloned - whichever is greater.
 pub_screen_name: The name of the public screen to clone.  e.g. "Workbench".
    sScreenTitle: A pointer to a null terminated character string to
                  appear in the screens title bar.  The Gui doesn't make a
                  copy of this string, it uses the pointer supplied so you
                  should make sure that this pointer remains valid while
                  the screen is open.
       LastWinFn: See
                 OpenGuiScreen
                 for details.
           flags: The GS_CLONEFONT flag causes the public screens font to be
                  used for your new screen.  The GS_CLONEPENS flag causes
                  the public screens pens to be used for the new screen.
    new_pub_name: A pointer to a NULL terminated character string to use
                  as the screen's public name.  If PubName is NULL or "" or
                  the application is running under an OS version prior to
                  V36 then the screen will be private (Public screens
                  weren't available before V36), otherwise an attempt will
                  be made to make the new screen public with the specified
                  name.  Specifying a PubName on an OS version prior to V36
                  will not cause the function to fail, the PubName will
                  simply be ignored.
    OverScanType: See
                 OpenGuiScreen
                 for details.
       extension: This is for future expansion.  Always set this to NULL.

Returns:

   If successful, a pointer to a valid GuiScreen structure is returned.  If
   not then NULL is returned.

Known bugs:

   None.

See also:


                 Destroy

                 GetNextAvailableDisplayMode
```

OpenGuiScreen

ScreenColoursFromILBM

ShowDisplayList

1.40 GetScreenDetails function

Function prototype:

```
struct Screen *GetScreenDetails(void *scr, unsigned long *mode, int *depth,  ↩
    char *fontname, int bufsize,
   int *reqbufsize, int *fontheight, int *fontstyle, UWORD *pens, int  ↩
      pensarraysize)
```

Description:

Get the details of a currently open public or FoxGUI screen. Requires V36.

Parameters:

```
            scr: Pointer to the FoxGUI screen or name of public screen.
           mode: Returns the screen mode of the specified screen.  Set to
                 NULL if you don't need to know the display mode.
          depth: Returns the depth of the specified screen.  Set to NULL
                 if you don't need to know the screen depth.
       fontname: Pointer to a character array to hold the returned name of
                 the screens font.  Set this to NULL if you don't need to
                 known the font name.
        bufsize: The size of the fontname string.
     reqbufsize: Returns the required size for the fontname string.  If
                 after calling GetScreenDetails, the reqbufsize returned
                 is larger than the size of the fontname string then the
                 font name will be incomplete and you should allocate a
                 larger string and call this function again to get the
                 full font name.
     fontheight: If this is not NULL then the size of the font for the
                 specified screen is returned here.
      fontstyle: If this is not NULL then the style of the font for the
                 specified screen is returned here.
           pens: An array to contain the pens array for the specified
                 screen.  Unless you know the number of pens used by the
                 screen then it is safest to make sure that this size of
                 this array is NUMDRIPENS.  Pass NULL if you don't need to
                 know the screen pens.
intpenarraysize: The size of your pens array.
```

Returns:

A pointer to the intuition screen struvture for the specified screen. Note that if the specified screen was a public non-FoxGUI screen then the screen will not be locked at this point so the screen could close at

```
   any time and this pointer could become invalid.

Known bugs:

   None.

See also:

                Destroy

                GetNextAvailableDisplayMode

                OpenGuiScreen

                ScreenColoursFromILBM

                ShowDisplayList
```

1.41 OpenGuiScreen function

```
               Function prototype:

   GuiScreen *OpenGuiScreen(int Depth, int DPen, int BPen, char *Title, int
      (*LastWinFn)(GuiScreen *), int flags, char *PubName,
      unsigned long DisplayID, int OverscanType, UWORD *pens, void *extension)

Description:

   Open a new screen with the attributes specified.

Parameters:

     Depth: The number of planes for the screen.  This determines the number
            of colours that can be shown on the screen (i.e.  1 plane gives 2
            colours, 2 planes give four colours, 3 planes give 8 colours
            etc).
      DPen: The detail pen colour for the screen.
      BPen: The block pen colour for the screen.
     Title: A pointer to a null terminated character string to appear in the
            screens title bar.  The Gui doesn't make a copy of this string,
            it uses the pointer supplied so you should make sure that this
            pointer remains valid while the screen is open.
 LastWinFn: For public screens, this can point to a function which you
            want to be triggered when the last visiting window (i.e.
            window opened by an application other than the current one) on
            your public screen is closed.  If you wish, you can use the
            same function for more than one screen if your application has
            more than one.  You can differentiate between the screens by
            using the GuiScreen pointer which is automatically passed to
            the function.  The function should return either
                 GUI_END
                 or
```

```
     GUI_CONTINUE
    .  If for example you wanted your application to
finish as soon as the user closed the last window on the
screen, you could do something like this:

int CALLBACK MyLastWinFunction(GuiScreen *scr)
{
   /* All visiting windows are now closed.  Check to see
      whether any of my own windows are open. */
   BOOL AllMyWindowsAreClosed = AllClosed();

   if (AllMyWindowsAreClosed)
      return GUI_END;
   else
      return GUI_CONTINUE;
}

If you do not want a function triggered then pass NULL for
LastWinFn.  If PubName is NULL or "" or if the application is
running on an OS version prior to V36 then LastWinFn is
ignored.
    flags: The following flags are available for screens: GS_AUTOSCROLL,
GS_INTERLACE, GS_DISPLAY_ID, GS_OVERSCAN and GS_PENS.
GS_AUTOSCROLL specifies that if the screen width and height are
greater than the display width and height then the screen will
automatically scroll to reveal the rest of the screen whenever
the mousepointer is brought close to the edge of the display.
At the moment, OpenGuiScreen always opens the screen so that it
fits the display so this flag is not very useful at the moment.
The GS_INTERLACE flag specifies that the screen should be
interlaced - this gives the screen twice as many rows (i.e.
doubles the screen's vertical resolution) but can cause the
display to be very flickery - especially on a TV screen.
GS_INTERLACE will be ignored if the GS_DISPLAY_ID flag is used
to describe the screenmode.
  PubName: A pointer to a NULL terminated character string to use as
the screen's public name.  If PubName is NULL or "" or the
application is running under an OS version prior to V36 then
the screen will be private (Public screens weren't available
before V36), otherwise an attempt will be made to make the new
screen public with the specified name.  Specifying a PubName on
an OS version prior to V36 will not cause the function to fail,
the PubName will simply be ignored.
DisplayID: An Intuition Display ID describing the screen mode for the
new screen.  This parameter is ignored unless the
GS_DISPLAY_ID flag was specified.  You should make sure that
the DisplayID is valid by looking it up in the display database
first.  This can be done using standard Intuition functions or
by calling the FoxGUI functions
     GetNextAvailableDisplayMode
                or
     ShowDisplayList
    OverScanType: The overscan style to use for the new screen.  This ↩
       should
have one of the values OSCAN_TEXT, OSCAN_STANDARD, OSCAN_MAX or
OSCAN_VIDEO but will be ignored unless the GS_OVERSCAN flag was
specified.  These values are defined by Intuition and you
```

```
            should see the Rom Kernel Reference Manuals for more
            information.
      Pens: A pointer to a Pen array for the new screen.  This is an
            array of pen colours terminated by the value ~0.  This
            parameter will be ignored unless the GS_PENS flag was
            specified.  If the GS_PENS flag is not specified then, if the
            application is used on an Amiga that does not support the
            three-d look, the Detail pen and Block pen specified in the
            DPen and BPen parameters will be used or, if the application is
            run on an Amiga which does support the three-d look then an
            attempt will be made to use a copy of the Workbench pens.  If
            that attempt fails, then the default pen set will be used.
 extension: This is for future expansion.  Always set this to NULL.

Returns:

   If successful, a pointer to a valid GuiScreen structure is returned.  If
   not then NULL is returned.

Known bugs:

   None.

See also:


                ClonePublicScreen

                Destroy

                GetNextAvailableDisplayMode

                ScreenColoursFromILBM

                ShowDisplayList
```

1.42 GetModeName function

```
               Function prototype:

   int GetModeName(unsigned long displaymode, char *buffer, int buflen);

Description:

   Looks up the name of the specified display mode ID in the display
   database and copies the name into the buffer supplied.  If the buffer is
   not long enough to store the whole mode name, as much as possible is
   copied into the buffer.

Parameters:

   displaymode: The displaymode whose name you want to find.
        buffer: A pointer to a buffer to store the name.
        buflen: The length of the buffer.
```

Returns:

The buffer length required to store the name of the specified mode.

Known bugs:

None.

See also:

GetModeSize

GetNextAvailableDisplayMode

ShowDisplayList

1.43 GetModeSize function

Function prototype:

```
BOOL GetModeSize(unsigned long displaymode, long *width, long *height);
```

Description:

Returns the width and height of a display mode. Works for most modes present on earlier Amigas and all modes on Amigas which support the display database.

Parameters:

```
displaymode: The mode id of the mode whose size you want to know.
      width: Returns the width of the requested mode.
     height: Returns the height of the requested mode.
```

Returns:

TRUE for success, FALSE otherwise.

Known bugs:

None.

See also:

GetModeName

GetNextAvailableDisplayMode

OpenGuiScreen

ShowDisplayList

1.44 GetNextAvailableDisplayMode function

```
                Function prototype:

   unsigned long GetNextAvailableDisplayMode(unsigned long previous);

Description:

   Gets the next available display mode supported by the Amiga running the
   software (the modes available will depend upon the version of the custom
   chips in the Amiga, the monitors installed and whether or not the Amiga
   has a graphics card).  The value returned from a call to this function
   can be passed to the
                OpenGuiScreen
                function to determine the display
   mode used by the new screen.

Parameters:

   previous: The value returned by the previous call to this function or
             INVALID_ID if this is the first call to this function.  A list
             of available mode ID's can be generated by calling the
             function passing INVALID_ID to get the first available mode
             then calling the function again passing the value returned the
             previous time to get the next one and so on.  When there are
             no more valid display mode ID's to return, the function will
             return INVALID_ID.  INVALID_ID is defined in graphics/modeid.h

Returns:

   The ID of the next available display mode.

Notes:

   Unlike the intuition functions for handling mode ID's the FoxGUI
   functions work on all Amigas.  On a basic Amiga 500 this function will
   return 4 valid mode ID's before returning INVALID_ID.  The ID's will be
   for the modes LORES, HIRES, LORES Laced and HIRES Laced.

See also:


                OpenGuiScreen

                ShowDisplayList
```

1.45 ShowDisplayList function

Function prototype:

```
BOOL ShowDisplayList(void *Scr, char *title, int DPen, int BPen,
   unsigned long *displayModeID);
```

Description:

Opens a window which contains a list box showing the names of all of the display modes available on the Amiga on which the program is run. The window also has Okay and Cancel buttons. The Okay button is not enabled until the user selects a mode from the list. The idea is that if you want the user of your program to be able to select the screen mode in which to open a screen you can call this function. The mode ID returned from a call to this function can be passed to the
OpenGuiScreen
function to determine the display mode used by the new screen.

Parameters:

Scr: A pointer to the screen on which to open the window.
title: A text string to be shown as the window title.
DPen: The detail pen of the window (see
OpenGuiWindow
).
BPen: The block pen of the window (see
OpenGuiWindow
).
displayModeID: A pointer to a variable to receive the selected display mode ID.

Returns:

TRUE if the user selected a mode and then pressed the Okay button, FALSE if the user pressed Cancel. If the user pressed Okay, the mode ID of the selected mode is returned in the displayModeID parameter.

Notes:

Unlike the intuition functions for handling mode ID's the FoxGUI functions work on all Amigas. On a basic Amiga 500 this function will show 4 valid mode ID's - LORES, HIRES, LORES Laced and HIRES Laced.

See also:

GetNextAvailableDisplayMode

OpenGuiScreen

1.46 FoxGUI Window functions

FoxGUI windows are standard Intuition windows made easy. I'm not ↩
going to

launch into a conceptual overview of what windows are and what you can do with them because if you're about to write an Amiga application then you're bound to know already and if you don't then you should probably read something else first. You can open one with a simple call to

OpenGuiWindow

and there are plenty of other windows functions below to make your programming life very simple.

The following window functions are currently available :-

OpenGuiWindow

SetFName

SetPath

SetWindowLimits

ShowFileRequester

SleepPointer

UpdateFList

WakePointer

WinBlankToEOL

WinClear

WinHideCursor

WinHome

WinPrint

WinPrintCol

WinPrintTab

WinShowCursor

WinTab

WinWrapOff

WinWrapOn

See also:

Destroy

1.47 OpenGuiWindow function

```
                Function prototype:

   GuiWindow *OpenGuiWindow(void *Scr, int Left, int Top, int Width, int Height,
      int Dpen, int Bpen, char *Title, int flags,
      int (*eventfn)(GuiWindow*, int, int, int, void*), void *extension)

Description:

   Open a new FoxGUI window on the specified screen.

Parameters:

         Scr: A pointer to an open FoxGUI screen in which the window is to
              be opened or a text string containing the name of a public
              screen on which to open the window.  If Scr is the name of a
              public screen and the application is run on an Amiga which
              does not support public screens or if the public screen does
              not exist the function will return NULL.
        Left: The coordinate of the left edge of the window relative to
              the left edge of the screen.
         Top: The coordinate of the top edge of the window relative to the
              top edge of the screen.  Note that this is from the very top
              of the screen so a y coordinate of 0 will cause the window to
              at least partly obscure the screens title bar if it has one.
       Width: The width of the window in pixels.
      Height: The height of the window in pixels.
        Dpen: The detail pen colour for the window.
        Bpen: The block pen for the window.
              The detail and block pen colours are passed directly to the
              intuition OpenWindow() function.  When your application is
              run on OS2.0 or higher these parameters are largely ignored
              because intuition uses the screen's pen colours to create the
              3D look for the window border.  On earlier OS releases, these
              two parameters are used to determine the window colours.
       Title: A pointer to a NULL terminated text string to appear in the
              window's title bar.  If you set this to NULL or "" then the
              window will have no title.  Setting this to "" will force the
              window to have a title bar whereas NULL will allow the window
              to have no title bar at all (specifying some of the gadget
              flags such as GW_DRAG and GW_CLOSE will also force the window
              to have a title bar).  If the Title is non-NULL then it is
              your responsibility to ensure that the text string remains
              valid - OpenGuiWindow does not make it's own copy of the
              string.
       flags: The following flags are currently available for Gui windows :-
              GW_CONSOLE, GW_DRAG, GW_BORDERLESS, GW_DEPTH, GW_CLOSE,
              GW_SIZE, GW_BACKDROP, GW_ACTIVE, GW_DROP, GW_DISKIN and
              GW_DISKOUT.  You can select more than one of these by ORing
              them together e.g. GW_DRAG | GW_DEPTH.

              GW_CONSOLE causes a console to be opened in the window.  This
              allows you to easily write text into the window using the
              following console functions :-
```

```
   WinBlankToEOL

   WinPrint

   WinShowCursor

   WinClear

   WinPrintCol

   WinTab

   WinHideCursor

   WinWrapOff

   WinHome

   WinPrintTab

   WinWrapOn
                Note that Intuition places a limit on the number of consoles ↩
that can be open at a time which is currently 4.  If you
specify the GW_CONSOLE flag and the application can't open
another console then OpenGuiWindow will fail and return NULL.

GW_BACKDROP causes the window to be a backdrop window.
Backdrop windows have no imagery and always appear behind all
non-backdrop windows on the screen.  Backdrop windows cannot
have close, size or depth gadgets and are never draggable so
if GW_BACKDROP is specified, the GW_CLOSE, GW_SIZE, GW_DEPTH
and GW_DRAG flags, if specified will be ignored.

GW_DRAG causes the window to be dragable (i.e.  you can drag
the window around the screen by pressing the left mouse
button when the pointer is over the window's title bar and
then keeping the button pressed down while you drag the mouse
around the screen).  If you specify this flag, your
event function (if you specify one - see eventfn below)
will be called whenever the user drags the window.

GW_BORDERLESS causes the window to have no border.  If any of
the gadget flags are specified (such as GW_CLOSE) or the
window has a title then the title bar will still be drawn but
the border around it and around the other edges of the window
will not be shown.

GW_DEPTH will create a depth-gadget at the right hand end of
the window's title bar.  Clicking this when the window is
partly obscured by other windows will bring the window to the
front.  If the window is already at the front then clicking
the depth gadget will send the window to the back.  On older
versions of the Amiga OS, two gadgets are created - one for
each of these functions.

GW_SIZE causes the window to have a size gadget at the bottom
```

of the right hand border. This allows the user to size the window. The minimum and maximum sizes of the window can be specified by calling the function
SetWindowLimits
. If
controls within the window are created with the
S_AUTO_SIZE
flag set then they will automatically get moved and ↩ resized
when the window is resized. For best results, open your window as small as possible so that the controls all fit in and are nicely spaced and set minwidth and minheight equal to the width and height parameters respectively. Making the window bigger will make the controls proportionately bigger too. If you specify this flag, your event function (if you specify one - see eventfn below) will be called whenever the user drags the window.

GW_CLOSE will create a close gadget in the left hand end of the window's title bar. You can control the operation of the close gadget by supplying your own event function (see the eventfn parameter) or, if you don't supply an event function, FoxGUI will just close the window for you when the close gadget is clicked.

GW_ACTIVE specifes that you want your application to know whenever the window becomes activated (i.e. when the user clicks in the window when another window has been active). Note that this event will usually occur when you first create a window. If you specify this flag, you will need to specify a function to call when the window becomes active. See the eventfn parameter below.

GW_DROP specifies that the window is allowed to have items "dropped" into it when writing an application which has Drag and Drop functionality. If the GW_DROP flag is specified then your event function will be called whenever something is dropped in your window (see the eventfn parameter below).

GW_DISKIN causes your window to be notified whenever a disk is inserted. If you specify this flag, you will need to specify a function to call when a disk is inserted (see the eventfn parameter below).

GW_DISKOUT causes your window to be notified whenever a disk is removed. If you specify this flag, you will need to specify a function to call when a disk is removed (see the eventfn parameter below).

eventfn: This is a pointer to a function to be called whenever events occur in your window. The event that occurred will be passed as a parameter to the function along with other details such as the x and y coordinates of the event if applicable. The event function will only be called for events which you have specified in the flags parameter. If you have not specified any events in the flags parameter (or if you do not wish to do any processing on those events) then this parameter can be

NULL. If you have specified the flag GW_CLOSE and this parameter is NULL then FoxGUI will simply close your window if the close button is clicked. Your event function should have the following prototype:

```
int CALLBACK Eventfn(GuiWindow *WhichWindow, int
   Event, int x, int y, void *DropData);
```

As you can see, the function will be passed a pointer to the window in which the event occurred, thus allowing you to use one function to handle events for more than one window. Your function should return either

```
   GUI_CONTINUE
   or
   GUI_END
  .
```

If the event is GW_CLOSE you can also optionally return GUI_CANCEL ORed with either GUI_CONTINUE or GUI_END. If you return GUI_CANCEL then FoxGUI will not close the window for you, otherwise it will. If your event function for the GW_CLOSE event closes the window itself then you must return GUI_CANCEL otherwise FoxGUI will attempt to close the already closed window and a crash is likely.

Depending on the flags you have specified, your event function may also be called whenever an object is dropped in the window (in which case the event will be GW_DROP) or the window is sized (GW_SIZE), dragged (GW_DRAG) or activated (GW_ACTIVE) or whenever a disk is inserted (GW_DISKIN) or removed (GW_DISKOUT).

The parameters to your event function are described below.

WhichWindow is a pointer to the GuiWindow affected. This allows you to use the same event function for more than one GuiWindow if you wish.

Event will be either GW_CLOSE, GW_DROP, GW_SIZE, GW_DRAG, GW_ACTIVE, GW_DISKIN or GW_DISKOUT depending on what just happened to your window.

If the event is GW_DROP then x and y will contain the coordinates of the point in the window that the object was dropped. If the event is GW_DRAG then x and y will contain the new coordinates of the top left corner of the window relative to the screen. If the event is GW_SIZE then x and y will contain the new width and height of the window. If the event is GW_ACTIVE then x and y will be zero. The values of x and y are unspecified if the event is GW_DISKIN, GW_DISKOUT or GW_CLOSE.

If the event is GW_DROP then DropData will be a pointer to the data which was initialised in the drag event. Otherwise DropData will be NULL. Unlike the event function for a Frame, the event function for a window receives a pointer to the dragged data (rather than a pointer to a pointer to the dragged data). This is because you cannot drag data out of a

window - only into it and so you will never need to modify the
data pointer from within a window's drop function. For
further information on drag and drop see the
MakeFrame
function.

extension: This parameter is for future extension and should be NULL.

Returns:

If successful, a pointer to the new FoxGUI window is returned.
Otherwise NULL is returned.

Known bugs:

When the GW_CONSOLE flag is specified, no console is currently created.
I hope to have this fixed in the next release. In the meantime, do not
use any of the console functions.

See also:

SetWindowLimits

Drag/Drop functionality

Destroy

ShowFileRequester

SleepPointer

WakePointer

1.48 SetFName function

Function prototype:

```
void SetFName(char *fname);
```

Description:

Set the file name in an open, non ASL file requester created by calling
the function
ShowFileRequester
. You would typically use this to reset
the file name if the user had attempted to open a file that didn't
exist. Note that if your application is running on an Amiga with ASL
library version 37 or greater then ShowFileRequester will open an ASL
file requester (the standard Amiga file requester). In this case, this
function will do nothing.

Parameters:

```
   fname: A pointer to a NULL terminated text string containing the file
          name to put in the file requester.  It is not necessary to
          maintain the string after calling the function because FoxGUI
          will make it's own copy of the string.

Known bugs:

   None.

See also:

                SetPath

                ShowFileRequester

                UpdateFList
```

1.49 SetPath function

```
              Function prototype:

   void SetPath(char *path);

Description:

   Set the file path in an open, non ASL file requester created by calling
   the function
                ShowFileRequester
               .  You would typically use this to reset
   the file path if the user had attempted to open a file that didn't
   exist.  Note that if your application is running on an Amiga with ASL
   library version 37 or greater then ShowFileRequester will open an ASL
   file requester (the standard Amiga file requester).  In this case, this
   function will do nothing.

Parameters:

   path: A pointer to a NULL terminated text string containing the file
         path to put in the file requester.  It is not necessary to
         maintain the string after calling the function because FoxGUI
         will make it's own copy of the string.

Known bugs:

   None.

See also:

                SetFName

                ShowFileRequester
```

UpdateFList

1.50 ShowFileRequester function

Function prototype:

```
BOOL ShowFileRequester(GuiWindow *Wnd, char *path, char *fname, char *pattern, ↩
    char
   *title, BOOL Save, int (*callfn) (char*, char*))
```

Description:

Show a file requester. On Amigas with ASL library version 37 or above, this will open a standard ASL file requester, allowing the user to select the filename and path of a file to perform some specific action on (usually loading or saving the file). On Amigas without the ASL library a functionally equivalent file requester is opened which allows a file name and path to be selected in exactly the same way.

Parameters:

Wnd: A file requester must be "attached" to a window. Wnd should be a pointer to an open FoxGUI window to which the file requester will be attached. Any open FoxGUI window will do - the window will not be changed in any way. The only criterion to use when deciding which window to pass is which screen you want the file requester to appear in. The file requester will always appear on the same screen as the window that you attach it to.

path: The initial file path for the file requester.

fname: The default filename to show in the requester.

pattern: This allows you to filter the files that appear in the file list. Directories will always be shown. If you want all files to be shown then set the pattern to "" (never set this to NULL). If the pattern is not blank then the last characters of the filename must match the pattern e.g. if the pattern is set to "fox" then only file names ending "fox" will be shown. You would usually use this to show files of a certain type, e.g. to only show AmigaGuide files you might set this to ".guide". Note that the pattern is not case sensitive so ".guide" would also match ".GUIDE" or ".Guide" etc.

title: The title to appear in the top border of the file requester.

Save: If TRUE then the left-most button on the file requester will have the caption "Save" otherwise it will have the caption "Load". The colours of the file list are also inverted for Save file requesters.

callfn: A pointer to a user-defined function to call when the Save or Load button is pressed. Your function should have the following prototype:-

```
int MyFileFn(char *FName, char *FPath);
```

Your function will be passed the filename and file path of the file selected by the user in the parameters FName and FPath respectively. Note that there is no guarantee that the path

or the file exists because the user can type directly into the filename and path edit boxes in the file requester as well as select files from the file list. You should not modify the strings passed to this function as they are required by FoxGUI - make your own copy and modify that if necessary.

Your function should return either GUI_MODAL_END or GUI_CONTINUE
.
If GUI_MODAL_END is returned, the file requester will go away immediately. If GUI_CONTINUE is returned, the file requester will remain present. You would typically return GUI_CONTINUE if your function failed for some reason and GUI_MODAL_END otherwise.

Returns:

TRUE for success, FALSE for failure.

Known bugs:

None.

Other notes:

Prior to release 5.0, this function took many extra parameters - the colours for the file requester window. These parameters should be ommitted from 4.3 onwards where the window colours are based on the colours of the GuiWindow passed as the first parameter.

I originally implemented this file requester without the ASL library. It was designed to remain shown until you press the "Done" button. The idea was that you selected a file to load (or save), pressed the "Load" (or "Save") button and while your user-defined function did the necessary loading or saving, the file requester would remain shown but temporarily disabled. The ASL file requester behaves slightly differently - it goes away as soon as the Load or Save button is pressed. I didn't want this to happen because occasionally, when loading, the user may mistype a file name or saving may fail due to a floppy being write protected or the disk being full and I thought the file requester should still be there so that they could try again and should only disappear when the user clicked on the Done button. For this reason, I have implemented this in such a way that if the ASL file requester is used, it disappears during loading or saving (I have no control over that) but it reappears once loading/saving is complete if (and only if) your load/save function returns GUI_CONTINUE. If your load/save function returns GUI_MODAL_END then the file requester will go away.

See also:

SetFName

SetPath

UpdateFList

1.51 SleepPointer function

Function prototype:

```
BOOL SleepPointer(GuiWindow *win);
```

Description:

Puts the specified window to sleep (the mouse pointer becomes the stop-clock whenever the specified window is active. All controls in the window become inactive). The window can be woken again with the function

WakePointer

.

Parameters:

win: A pointer to an open FoxGUI window.

Returns:

TRUE for success, FALSE for failure.

Known bugs:

Due to a bug in earlier versions of the Amiga OS, it is impossible to detect (on early Amigas) whether the method that this function uses to disable the window has actually succeeded so on earlier Amigas this function always returns TRUE. On later Amigas the return value is completely reliable. In practice I have never known the window disabling to fail so this is unlikely to cause problems.

See also:

WakePointer

1.52 UpdateFList function

Function prototype:

```
void UpdateFList(void);
```

Description:

Forces an open file requester to update it's file list. This function does nothing on Amigas which use the ASL file requester (see the

ShowFileRequester

```
                function).  If no file requester is open, this
   function does nothing.

Known bugs:

   None.

See also:


                SetFName

                SetPath

                ShowFileRequester
```

1.53 WakePointer function

```
               Function prototype:

   void WakePointer(GuiWindow *win);

Description:

   Wake up a window previously put to sleep by the
                SleepPointer
                  function.  If the window is not asleep then this function does ↩
                     nothing.

Parameters:

   win: A pointer to the FoxGUI window to wake up.

Known bugs:

   None.

See also:


                SleepPointer
```

1.54 SetWindowLimits function

```
               Function prototype:

   BOOL SetWindowLimits(GuiWindow *gw, long minwidth, long minheight, unsigned ↩
      long maxwidth,
      unsigned long maxheight);
```

Description:

Set the minimum and maximum width and height of a resizeable window. If the window is not resizeable (i.e. wasn't created with the GW_SIZE flag) then this function will do nothing.

Parameters:

gw: A pointer to the FoxGUI window.
minwidth: The new minimum width of the window. If the window is currently not as wide as this value then it will be ignored. If this value is zero then the minimum width will remain unchanged.
minheight: The new minimum height of the window. If the window is currently not as high as this value then it will be ignored. If this value is zero then the minimum height will remain unchanged.
maxwidth: The new maximum width of the window. If the window is currently wider than this value then it will be ignored. If this value is zero then the maximum width will remain unchanged.
maxheight: The new maximum height of the window. If the window is currently higher than this value then it will be ignored. If this value is zero then the maximum height will remain unchanged.

Returns:

TRUE for success, FALSE for failure.

Known bugs:

None.

See also:

OpenGuiWindow

1.55 WinBlankToEOL function

Function prototype:

```
void WinBlankToEOL(GuiWindow *w)
```

Description:

Blank from the current cursor position to the end of the current line in the console attached to the specified window. The cursor can be moved with the
WinTab
function.

Parameters:

```
   w: A pointer to an open FoxGUI window which has a console attached.
See also:

                OpenGuiWindow

                WinClear

                WinHideCursor

                WinHome

                WinPrint

                WinPrintCol

                WinPrintTab

                WinShowCursor

                WinTab

                WinWrapOff

                WinWrapOn
```

1.56 WinClear function

```
               Function prototype:

   void WinClear(GuiWindow *w)
Description:

   Clear the console in the specified FoxGUI window.
Parameters:

   w: A pointer to an open FoxGUI window which has a console attached.
See also:

                OpenGuiWindow

                WinBlankToEOL

                WinHideCursor

                WinHome

                WinPrint
```

WinPrintCol

WinPrintTab

WinShowCursor

WinTab

WinWrapOff

WinWrapOn

1.57 WinHideCursor function

Function prototype:

```
void WinHideCursor(GuiWindow *w)
```

Description:

Hides the cursor in a FoxGUI window which has a console attached. By default the cursor is visible and at the top left corner of the window. The cursor can be moved by the WinTab function and by functions which print text in the console such as WinPrint. After being hidden, the cursor can be shown again with the function WinShowCursor.

Parameters:

w: A pointer to an open FoxGUI window which has a console attached.

See also:

OpenGuiWindow

WinBlankToEOL

WinClear

WinHome

WinPrint

WinPrintCol

WinPrintTab

WinShowCursor

WinTab

WinWrapOff

WinWrapOn

1.58 WinHome function

Function prototype:

```
void WinHome(GuiWindow *w)
```

Description:

Moves the cursor to the top left of the specified FoxGUI window which has a cursor attached. Calling WinHome(MyWindow) is entirely equivalent to calling WinTab(MyWindow, 1, 1).

Parameters:

w: A pointer to an open FoxGUI window which has a console attached.

See also:

OpenGuiWindow

WinBlankToEOL

WinClear

WinHideCursor

WinPrint

WinPrintCol

WinPrintTab

WinShowCursor

WinTab

WinWrapOff

WinWrapOn

1.59 WinPrint function

Function prototype:

```
void WinPrint(GuiWindow *w, char *str)
```

Description:

Prints the specified text string at the current cursor position in the console attached to the specified open FoxGUI window. If the text is too long to fit between the current cursor position and the end of the current line in the console then the text will either be truncated or the text will continue at the beginning of the next line of the console. Whether or not the text is truncated depends on whether text wrapping is turned on in the console. Text wrapping can be turned on and off with the functions
WinWrapOn
and
WinWrapOff
respectively. If the text reaches the bottom right hand corner of the window and text wrapping is turned on then the console will scroll up a line to allow room for the next line of text. This will remove the top line of text from the window. After printing the text, the cursor will be positioned immediately after the last character that was printed unless the right hand edge of the console has been reached with text wrapping turned off in which case the cursor will be on the last character printed.

Parameters:

w: A pointer to an open FoxGUI window which has a console attached.
str: The text that you want to be printed at the current cursor position in the specified windows console.

See also:

OpenGuiWindow

WinBlankToEOL

WinClear

WinHideCursor

WinHome

WinPrintCol

WinPrintTab

WinShowCursor

WinTab

WinWrapOff

WinWrapOn

1.60 WinPrintCol function

```
                Function prototype:

   void WinPrintCol(GuiWindow *w, char *str, int col)

Description:

   This function prints the text string specified into the console of the
   specified open FoxGUI window (the window must have a console attached at
   the time it is opened) in the specified colour.  The way the text is
   printed will depend on whether text wrapping is currently turned on or
   off in the console.  See
                WinPrint
                for a full explanation.

Parameters:

     w: A pointer to an open FoxGUI window which has a console attached.
   str: The text that you want to be printed at the current cursor position
        in the specified windows console.
   col: The colour in which the text should be printed.  This should be an
        integer whose maximum value will be dependant on the number of
        bitplanes used by the screen.  For example, if the screen is opened
        with one bitplane then col should be between 0 and 1.  If the
        screen has two bitplanes then col should be between 0 and 3.  In
        general, col should be between 0 and 2^n - 1 where n is the number
        of bitplanes.

See also:


                OpenGuiWindow

                WinBlankToEOL

                WinClear

                WinHideCursor

                WinHome

                WinPrint

                WinPrintTab

                WinShowCursor

                WinTab

                WinWrapOff

                WinWrapOn
```

1.61 WinPrintTab function

```
                Function prototype:

   void WinPrintTab(GuiWindow *w, int x, int y, char *str)

Description:

   This function first moves the console cursor of the specified open
   FoxGUI window to the coordinates given in x and y and then prints the
   specified text string at that position.  Calling this function is
   equivalent to calling the
                WinTab
                function followed by the
                WinPrint
                  function and you should read the sections on those two  ↩
                     functions for further
   details.

Parameters:

     w: A pointer to an open FoxGUI window which has a console attached.
     x: The x coordinate to move the cursor to before printing the text.
     y: The y coordinate to move the cursor to before printing the text.
   str: The text that you want to be printed.

See also:


                OpenGuiWindow

                WinBlankToEOL

                WinClear

                WinHideCursor

                WinHome

                WinPrint

                WinPrintCol

                WinShowCursor

                WinTab

                WinWrapOff

                WinWrapOn
```

1.62 WinShowCursor function

```
                Function prototype:

   void WinShowCursor(GuiWindow *w)

Description:

   Shows the cursor in the specified window.  The window must have a
   console attached.  The cursor is shown by default so it is only
   necessary to call this function if you have previously hidden the cursor
   with the
                WinHideCursor
                function and you now want to make it visible
   again.

Parameters:

   w: A pointer to the open FoxGUI window whose console cursor you wish to
      show.

See also:


                OpenGuiWindow

                WinBlankToEOL

                WinClear

                WinHideCursor

                WinHome

                WinPrint

                WinPrintCol

                WinPrintTab

                WinTab

                WinWrapOff

                WinWrapOn
```

1.63 WinTab function

```
                Function prototype:

   void WinTab(GuiWindow *w, int x, int y)

Description:
```

```
   Move the specified windows console cursor to the coordinates specified.
   Console coordinates have the origin at (1,1).  i.e.  the top left of the
   console is at x=1, y=1.  Specifying 0 for either coordinate will just
   set that coordinate to 1.

Parameters:

   w: A pointer to an open FoxGUI window which has a console attached.
   x: The x coordinate to move the cursor to.
   y: The y coordinate to move the cursor to.

See also:

                OpenGuiWindow

                WinBlankToEOL

                WinClear

                WinHideCursor

                WinHome

                WinPrint

                WinPrintCol

                WinPrintTab

                WinShowCursor

                WinWrapOff

                WinWrapOn
```

1.64 WinWrapOff function

```
               Function prototype:

   void WinWrapOff(GuiWindow *w)

Description:

   Turn off text wrapping for the console in the specified FoxGUI window.
   For a description of how text wrapping affects the way text is printed
   in a console, see the
                WinPrint
                function.

Parameters:

   w: A pointer to an open FoxGUI window which has a console attached.
```

See also:

OpenGuiWindow

WinBlankToEOL

WinClear

WinHideCursor

WinHome

WinPrint

WinPrintCol

WinPrintTab

WinShowCursor

WinTab

WinWrapOn

1.65 WinWrapOn function

Function prototype:

```
void WinWrapOn(GuiWindow *w)
```

Description:

Turn on text wrapping for the console in the specified FoxGUI window. For a description of how text wrapping affects the way text is printed in a console, see the WinPrint function.

Parameters:

w: A pointer to an open FoxGUI window which has a console attached.

See also:

OpenGuiWindow

WinBlankToEOL

WinClear

WinHideCursor

WinHome

WinPrint

WinPrintCol

WinPrintTab

WinShowCursor

WinTab

WinWrapOff

1.66 FoxGUI Menu functions

Menus are attached to FoxGUI windows and appear in the title bar of the screen containing the window when the right mouse button is pressed down (but then, being an Amiga user you already knew that). FoxGUI allows a set of menus to be shared between multiple windows without having to create an identical set for each window (see ShareMenus below). The functions below should be obvious by their names.

The following menu functions are currently available :-

AddMenu

AddMenuItem

AddSubMenuItem

ClearMenus

DisableMenu

DisableMenuItem

DisableWinMenus

EnableMenu

EnableMenuItem

EnableWinMenus

IsMenuChecked

RemoveMenuItem

SetMenuChecked

SetWinMenuFn

ShareMenus

1.67 AddMenu function

Function prototype:

```
struct Menu *AddMenu(GuiWindow *win, char *name, int leftedge, int enabled);
```

Description:

Add a new top-level menu to the specified Gui window. Note that each gui window can have no more than 31 top-level menus. More than one Gui window can share the same set of menus - see ShareMenus Link ShareMenus}

Parameters:

win: A pointer to an open Gui window to which a new top-line menu will be added.
name: A pointer to a NULL-terminated text string which will appear in the gui window's menu bar.
leftedge: The offset of the new menu from the left hand edge of the screen in pixels.
enabled: If FALSE, the menu will initially be disabled. Otherwise enabled.

Returns:

If successful, a pointer to the new menu structure is returned. You will need to pass this to the function AddMenuItem when adding items to this top-line menu. If AddMenu fails, NULL is returned.

Known bugs:

None.

See also:

AddMenuItem

ClearMenus

DisableMenu

DisableWinMenus

EnableMenu

EnableWinMenus

SetWinMenuFn

ShareMenus

1.68 AddMenuItem function

Function prototype:

```
struct MenuItem *AddMenuItem(GuiWindow *win, struct Menu *menu,
   char *name, char *selname, unsigned short flags, int key, int enabled,
   int checkit, int checked, int menutoggle);
```

Description:

Adds a new menu item to the specified top-level menu in the specified window. There are three types of menu items - those that perform some user-defined action (these are known as Action items and have no special imagery), those that toggle between two states (these are known as Checkmark items and have a tick to the left of the item text which is either shown or hidden depending on their state) and those which contain sub-menus (these have the » symbol to the right of the text to indicate that a submenu is present). Shortcut keys can be applied to the first two types of menu. Menus containing sub-menus are made in exactly the same way as action menus. The sub-menu is then added using calls to the function

AddSubMenuItem

. Note that each top-level menu can contain no more than 63 menu items

Parameters:

win: A pointer to the open Gui window that the menu item is to be added to.

menu: A pointer to the top-level menu which this item is to appear in.

name: A pointer to a NULL-terminated text string to appear in the drop down menu.

selname: A pointer to an optional NULL-terminated text string to be displayed when the menu option is hilighted instead of the more usual hilighting method of inverting the pixels. Use NULL for normal hilighting.

flags: Currently unused but reserved for future enhancements. To ensure compatibility with later versions, set this to zero.

key: The shortcut key for the menu. For example, passing 'Q' would make right Amiga-Q the hotkey for the menu. Pressing right Amiga-Q would then have the same effect as selecting the menu. Pass 0 if no shortcut key is required.

enabled: If FALSE, the menu item will initially be disabled. Otherwise enabled.

checkit: If TRUE then this will be a Checkmark item.

checked: Set this to TRUE if the item is a Checkmark item and you want it to be initially checked (i.e. you want the tick to appear). If this is an action item or if it is a checkmark item which you want to be initially unchecked then set this

```
               to FALSE.
   menutoggle: Set this to TRUE if the item is a Checkmark item and you
               wish to be able to toggle the state by repeated selection of
               the item.  If this is FALSE and the item is a Checkmark
               item, the only way for the item to become un-checked is by
               mutual-exclusion which is not currently supported by FoxGUI.

Returns:

   If successful, a pointer to the new menu item is returned.  In the event
   of failure, NULL is returned.

Known bugs:

   None.

See also:


                 AddMenu

                 AddSubMenuItem

                 ClearMenus

                 DisableMenuItem

                 EnableMenuItem

                 SetWinMenuFn

                 ShareMenus
```

1.69 AddSubMenuItem function

```
                Function prototype:

   struct MenuItem *AddSubMenuItem(GuiWindow *win, struct MenuItem
      *menuitem, char *name, char *selname, unsigned short flags, int key,
      int enabled, int checkit, int checked, int menutoggle);

Description:

   Adds a sub-menu item to the specified existing menu item in the
   specified window.  Note that each menu item may have a maximum of 31
   sub-menu items.

Parameters:

   menuitem: A pointer to an item in an existing menu structure.  This item
             will become the parent item for the sub-menu item created.  If
             the parent item already has sub-menu items then the new one
             will be added to the end of the sub-menu, otherwise a new
             sub-menu will be created with this item being the first in the
```

```
            new sub-menu.  AddSubMenuItem will fail if the menuitem
            specified to be the parent is itself a sub-menu item as
            sub-sub-menu items are not supported.

   All other parameters are identical to the parameters of the function

                AddMenuItem
               .

Returns:

   If successful, a pointer to the new sub-menu item is returned.  In the
   event of failure, NULL is returned.

Known bugs:

   None.

See also:

                AddMenu

                AddMenuItem

                ClearMenus

                DisableMenuItem

                EnableMenuItem

                SetWinMenuFn

                ShareMenus
```

1.70 ClearMenus function

```
Function prototype:

   void ClearMenus(GuiWindow *win);

Description:

   Removes all top-level menus, menu items and sub-menu items from the
   specified Gui window.  If the menus are shared with other Gui windows,
   the other windows remain unaffected.  If the menus are not shared with
   other windows, all the resources used by the menus are released.  You
   should always call ClearMenus before closing a Gui window which has
   menus.

Parameters:

   win: The Gui window whose menus are to be cleared.
```

Known bugs:

None.

1.71 DisableMenu function

Function prototype:

```
BOOL DisableMenu(GuiWindow *win, struct Menu *menu);
```

Description:

Disables the specified top-level menu in the specified window and all other windows which share the same menu strip. This prevents the menu from being dropped and hence prevents selection of any of the menu items within it. The menu text will appear ghosted.

Parameters:

win: A pointer to the window containing the menu to be disabled.
menu: A pointer to the top-level menu to be disabled.

Returns:

TRUE for success, FALSE for failure.

Known bugs:

None.

See also:

DisableMenuItem

DisableWinMenus

EnableMenu

EnableMenuItem

EnableWinMenus

1.72 DisableMenuItem function

Function prototype:

```
BOOL DisableMenuItem(GuiWindow *win, struct MenuItem *item);
```

Description:

```
   Disables the specified menu-item in the specified window and all other
   windows that share the same menu strip.  The item text will appear
   ghosted and will become un-selectable.

Parameters:

   win:  A pointer to the Gui window containing the menu item.
   item: A pointer to the item to be disabled.

Returns:

   TRUE for success, FALSE for failure.

Known bugs:

   None.

See also:

                DisableMenu

                DisableWinMenus

                EnableMenu

                EnableMenuItem

                EnableWinMenus
```

1.73 DisableWinMenus function

```
               Function prototype:

   BOOL DisableWinMenus(GuiWindow *win);

Description:

   Disables all menus in the window specified and all other windows that
   share the same menu strip.

Parameters:

   win: A pointer to the Gui window whose menus are to be disabled.

Returns:

   TRUE for success, FALSE for failure.

Known bugs:

   None.

See also:
```

```
                DisableMenu

                DisableMenuItem

                EnableMenu

                EnableMenuItem

                EnableWinMenus
```

1.74 EnableMenu function

```
               Function prototype:

   BOOL EnableMenu(GuiWindow *win, struct Menu *menu);

Description:

   Enables the specified top-level menu in the specified window and all
   other windows which share the same menu strip.

Parameters:

    win: A pointer to the window containing the menu to be enabled.
   menu: A pointer to the top-level menu to be enabled.

Returns:

   TRUE for success, FALSE for failure.

Known bugs:

   None.

See also:

                DisableMenu

                DisableMenuItem

                DisableWinMenus

                EnableMenuItem

                EnableWinMenus
```

1.75 EnableMenuItem function

```
                Function prototype:

   BOOL EnableMenuItem(GuiWindow *win, struct MenuItem *item);

Description:

   Enables the specified menu-item in the specified window and all other
   windows that share the same menu strip.

Parameters:

   win:  A pointer to the Gui window containing the menu item.
   item: A pointer to the item to be enabled.

Returns:

   TRUE for success, FALSE for failure.

Known bugs:

   None.

See also:

                DisableMenu

                DisableMenuItem

                DisableWinMenus

                EnableMenu

                EnableWinMenus
```

1.76 EnableWinMenus function

```
                Function prototype:

   BOOL EnableWinMenus(GuiWindow *win);

Description:

   Enables all menus in the window specified and all other windows that
   share the same menu strip.

Parameters:

   win: A pointer to the Gui window whose menus are to be enabled.

Returns:

   TRUE for success, FALSE for failure.
```

Known bugs:

None.

See also:

DisableMenu

DisableMenuItem

DisableWinMenus

EnableMenu

EnableMenuItem

1.77 IsMenuChecked function

Function prototype:

```
BOOL IsMenuChecked(struct MenuItem *mi);
```

Description:

Determines whether or not the specified checkmark menu item is checked.

Parameters:

mi: A pointer to a menu item.

Returns:

TRUE if the specified menu item is checked, FALSE otherwise.

Known bugs:

None.

See also:

AddMenuItem

SetMenuChecked

1.78 RemoveMenuItem function

Function prototype:

```
BOOL RemoveMenuItem(GuiWindow *win, struct MenuItem *item);
```

Description:

Removes the specified menu item from it's parent menu in the specified window. If the menu strip for that window is shared with other windows then the item is removed from all of those windows. If the item has sub-menu items below it then that sub-menu will also be removed.

Parameters:

win: The window from which a menu item is to be removed.
item: A pointer to the menu item which is to be removed. If the item is removed successfully then this pointer will no-longer point to a meaningful structure and should be discarded.

Returns:

TRUE if the item was removed successfully, FALSE otherwise.

Known bugs:

Currently can't be used to remove a sub-menu item or a top-level menu. These can only be removed by clearing the menu strip using the

ClearMenus
function and creating the menus again.

See also:

AddMenuItem

ClearMenus

DisableMenu

1.79 SetMenuChecked function

Function prototype:

```
BOOL SetMenuChecked(GuiWindow *win, struct MenuItem *item, BOOL
   checked);
```

Description:

Sets the state of a checkmark menu item.

Parameters:

win: A pointer to the GuiWindow containing the menu item.

```
      item: A pointer to the menu item.
   checked: TRUE of you want the menu to become checked, FALSE otherwise.

Returns:

   TRUE for success, FALSE for failure.

Known bugs:

   None.

See also:


                AddMenuItem

                IsMenuChecked
```

1.80 SetWinMenuFn function

```
               Function prototype:

   void SetWinMenuFn(GuiWindow *win, int (*fn)
         (GuiWindow*, struct MenuItem *))

Description:

   Set or change the function to be invoked when an action menu in the
   specified window is selected.  This function allows the programmer to
   write his/her own function which will be called whenever the user
   selects an action menu in the specified window.  Windows that share
   menus do not necessarily have to have the same menu function so if you
   want two windows to share the same menu function it is necessary to call
   SetWinMenuFn once for each window.

Parameters:

   win: The Gui window whose menu function you wish to set.
    fn: A pointer to the menu function for the specified window.  The
        function should have the following prototype:
        int MyWindowMenuFunction(GuiWindow *WhichWindow, struct MenuItem * ↩
           WhichMenuItem);
        The function will be passed a pointer to the menu item that
        was selected and the window it was attached to.  It should return
        either
                 GUI_CONTINUE
                 or
                 GUI_END
                .

Known bugs:

   None.
```

See also:

AddMenu

OpenGuiWindow

ShareMenus

1.81 ShareMenus function

Function prototype:

```
void ShareMenus(GuiWindow *dest, GuiWindow *source);
```

Description:

Causes a set of menus to be shared between two or more windows. When a number of windows share the same menu strip, most FoxGUI menu functions which are applied to that menu strip will affect the menus in all of those windows (the only current exception being the function SetWinMenuFn which only affects the window passed as it's second parameter). For example, if windows A and B share the same menu strip and AddMenu is called to add a new top-level menu to window A, it will also be added to window B.

In order to make Window B share menus already created for Window A, call ShareMenus(B, A). If window B already has menus they will be cleared. In order to then use the same menus for a third window C you could either use ShareMenus(C, B) or ShareMenus(C, A).

Parameters:

dest: A pointer to the Gui window to which the menus will be applied.
source: A pointer to the Gui window whose existing menus are to be shared.

Known bugs:

The source window must have at least one top-level menu attached to it for ShareMenus to work. If it hasn't, the function won't cause them to share menus. As long as source has one top-level menu when ShareMenus is called, any other alterations to the menu structure of either window will affect both.

See also:

AddMenu

SetWinMenuFn

1.82 FoxGUI Button functions

Buttons are probably the most commonly used intuition "gadgets" so I've made them very simple. The functions below should all be obvious by their names. If not then you'll just have to read the function descriptions! Buttons can have images attached to them (drawn on them) - see the

AttachBitMapToControl function.

The following button functions are currently available :-

MakeButton

See also:

Destroy

DisableControl

EnableControl

SetDelay

SetPeriod

AttachBitMapToControl

1.83 MakeButton function

Function prototype:

```
PushButton *MakeButton(void *Parent, char *name, int left, int top, int
width, int height, int key, struct Border *cb, int
(*callfn) (PushButton*), short flags, void *extension)
```

Description:

Create a FoxGUI button in the specified window or frame with the attributes specified. The button will automatically be enabled and when the user clicks the button the function specified in callfn will be invoked and passed a pointer to the button that was clicked.

Parameters:

Parent: A pointer to an open FoxGUI window or frame in which to place the new button.

name: The text to appear as the buttons caption. If you don't want a caption, set this to "" or NULL. If you want the button to have a hot-key (a key that is used to activate the button from the keyboard) then you should pass the relevant character in the

"key" parameter. If that key is one of the characters of the buttons caption then you can preceed that character by an underscore ("_") and that character will be underlined in the button caption. For example, for an okay button you might pass "O_kay" in which case the k will be underlined. If the caption for the button is too long to fit it then it will be truncated - the caption will never extend beyond the button's border. If the button gets resized (due to a window being resized) then more of the caption may become visible - i.e. FoxGUI remembers the whole caption not just the visible bit.

left: The coordinate of the left edge of the button relative to the left edge of the window/frame. If you prefer, you can specify a negative value here and that will be taken to mean an offset of the left hand edge of the button from the right hand edge of the window/frame. If the window/frame gets resized, this will also cause the button to move to remain at the same offset from the right hand edge of the window/frame.

top: The coordinate of the top edge of the button relative to the top edge of the window/frame. Note that this is from the very top of a window so a y coordinate of 0 would cause the button to be at least partly obscured by the window's title bar if it had one. If you prefer, you can specify a negative value here and that will be taken to mean an offset of the top edge of the button from the bottom edge of the window/frame. If the window/frame gets resized, this will also cause the button to move to remain at the same offset from the bottom of the window/frame.

width: The width of the button in pixels.

height: The height of the button in pixels.

key: The button's hot key. For example, to make the hot key the letter k, pass 'k'.

cb: If you want your button to have some form of custom imagery not directly supported by MakeButton, you can create your own intuition border structure and pass a pointer to it as cb. This will be displayed as well as the standard button border and caption (if there is one). It's main use is to draw a simple picture on the button in place of a caption. The caption is ommitted by passing the name as "".

callfn: A pointer to the function to be called when the button is clicked. The function should have the following prototype:

```
int CALLBACK MyButtonFunction (PushButton*);
```

When it is invoked, it will be passed a pointer to the button that was clicked (this allows you to have one button function which handles all of your buttons if you wish. Of course you could alternatively have a different function for each button in which case the parameter would be redundant). It should return either

GUI_CONTINUE
or
GUI_END
.

flags: The following flags are available for buttons: BN_CLEAR, BN_AR, BN_STD, BN_OKAY, BN_CANCEL
S_AUTO_SIZE
and S_FONT_SENSITIVE.

If BN_CLEAR is set then the background colour of the button will be the same colour as the window that the button is created in and the bcol parameter will be ignored. If you want to attach an image to a button then the button must be created with this flag specified. Clear buttons are drawn more quickly than coloured buttons. If you don't specify the BN_CLEAR flag but the bcol parameter is the same as the background colour of the window then the button won't be treated as clear so it will take marginally longer to refresh. If buttons are not clear then the whole button area is refreshed when necessary. If they are clear then only the border is refreshed.
BN_OKAY and BN_CANCEL allow either the return key or the escape key respecively to be used as an extra hot-key for the button. An example of how you might use these would be some sort of preferences window where the user can modify several controls (eg tick boxes, drop-down list boxes, edit boxes etc) and then either cancel or accept the action. You might have two buttons in the window labelled "Okay" and "Cancel" which have O and C as their respective hot-keys but where the Okay button has BN_OKAY set and the Cancel button has BN_CANCEL set so that the preferences could also be accepted or cancelled by pressing return or escape.
If the S_FONT_SENSITIVE flag is specified then the buttons height and width are set according to the height and width of the text on the button and the height and width parameters are ignored. Buttons with this flag set do not resize when in a resizable window (although they do still move as necessary).
The other two flags are mutually exclusive. BN_STD specifies that this is a standard button, BN_AR specifies that it is an auto-repeating button. If you click on a standard button, the function is activated when the button is released. If the mousepointer is moved off the button before it is released then the function isn't called. An auto-repeat button is activated immediately that the button is clicked and is repeatedly activated while the button is held down. The delay between each activation of the button can be set by calling

SetDelay
and

SetPeriod
.

extension: This is for future expansion and should be set to NULL.

Returns:

If successful, a pointer to the new FoxGUI button is returned. If unsuccessful, NULL is returned.

Known bugs:

None.

See also:

Destroy

DisableControl

EnableControl

SetDelay

SetPeriod

AttachBitMapToControl

1.84 FoxGUI Boolean Gadget functions

The boolean gadgets supported by FoxGUI are tick-boxes and radio buttons.
A Tick-box is a small square button with a tick in it (or not as the case may be). You make the tick appear or disappear by clicking on the button. Usually a tick specifies that some option is turned on and lack of a tick means that it is turned off.

Radio buttons are another type of boolean gadget. These are grouped together and selecting one causes any other in the same group to become de-selected (i.e. they are mutually exclusive) like the buttons on the front of old radios if you are old enough to remember them!

The following boolean gadget functions are currently available :-

ActiveRadioButton

MakeRadioButton

MakeTickBox

SetTickBoxValue

TickBoxValue

See also:

Destroy

DisableControl

EnableControl

1.85 ActiveRadioButton function

Function prototype:

```
RadioButton *ActiveRadioButton(RadioButton *rb);
```

```
Description:

   Given a pointer to any FoxGUI radio button, this function returns a
   pointer to the currently selected radio button in that group.

Parameters:

   rb: A pointer to a FoxGUI radio button.

Returns:

   A pointer to the currently selected radio button in the same group as
   the radio button passed as a parameter.  If no radio button is selected
   in that group or if an error occurs, this function returns NULL.

Known bugs:

   None.

See also:

                MakeRadioButton
```

1.86 MakeRadioButton function

```
               Function prototype:

   RadioButton *MakeRadioButton(void *Parent, RadioButton *MutEx, int left,
      int top, int width, int height, int fillcol, int (*callfn)
      (RadioButton*), int flags, void *extension)

Description:

   Make a new FoxGUI radio button in the specified FoxGUI window or frame.

Parameters:

       Parent: A pointer to an open FoxGUI window or frame in which to
               create the new radio button.
        MutEx: A pointer to another radio button which you want to be
               mutually exclusive with this one (i.e. in the same group).
               When creating the first radio button in a group, set this
               parameter to NULL.  When creating subsequent radio buttons
               in the group, this parameter can point to any one of the
               radio buttons already created and the whole group will be
               mutually exclusive (i.e.  selecting any one member in the
               group will cause any other button in the group that was
               previously selected to become un-selected).
         left: The distance in pixels between the left edge of the
               window/frame and the left edge of the selectable part of the
               radio button.  Remember to leave enough room for the caption
               on either the left or right of the radio button itself.
```

top: The distance in pixels between the top edge of the window/frame and the top edge of the radio button.
width: The width of the selectable part of the radio button (in pixels).
height: The height of the radio button in pixels.
fillcol: When a radio button is selected, the centre of the button gets filled in the colour specified in this parameter. Unselected radio buttons are not filled.
callfn: A function to call when this radio button is selected by the user. The function will be passed a pointer to the radio button selected so if you like you can use the same function for all of the radio buttons in a group or even all of the radio buttons in an application. If you don't want to perform any special action immediately that the radio button is selected, you can pass NULL for this parameter. If you do specify a function, it should have the following prototype :-

```
int CALLBACK MyRadioButtonFn (RadioButton *WhichRadioButton);
```

and should return either
GUI_CONTINUE
or

GUI_END
.

flags: Currently, there are only two valid flags for radio buttons: BG_SELECTED causes the radio button to be the initially selected radio button in the group. Obviously you should only set this flag for one radio button in each group. Radio buttons will auto-size when their window is resized if the
S_AUTO_SIZE
flag is selected.
extension: This is reserved for future expansion and should be set to NULL.

Returns:

If successful, a pointer to the new radio button. If not, NULL.

Known bugs:

None.

See also:

SetPreText

SetPostText

ActiveRadioButton

Destroy

1.87 SetTickBoxValue function

```
                Function prototype:

   BOOL SetTickBoxValue(TickBox *tb, BOOL value);

Description:

   Set the current value of the specified tick box.

Parameters:

      tb: The tick box whose value you want to set.
   value: TRUE if you want the tick box to be ticked, FALSE otherwise.

Returns:

   TRUE if the function succeeds, FALSE otherwise.

Known bugs:

   None.

See also:

                 TickBoxValue

                 MakeTickBox
```

1.88 TickBoxValue function

```
                Function prototype:

   BOOL TickBoxValue(TickBox *tb);

Description:

   Find the current value of the specified tick box.

Parameters:

   tb: The tick box whose value you want to know.

Returns:

   TRUE if the tick box is ticked, FALSE otherwise.

Known bugs:

   None.

See also:
```

SetTickBoxValue

MakeTickBox

1.89 MakeTickBox function

Function prototype:

```
TickBox *MakeTickBox(void *Parent, int left, int top, int width, int height,
 int (*callfn) (TickBox*), int flags, void *extension)
```

Description:

Make a new tick box gadget.

Parameters:

Parent: The Gui window or frame in which the new tick box is to be created.
left: The left edge of the tick box in pixels from the left edge of the window/frame.
top: The top edge of the tick box in pixels from the top edge of the window/frame.
width: The width of the tick box in pixels.
height: The height of the tick box in pixels.
callfn: A function to call whenever the user changes the state of the tick box by clicking on it with the mouse. The prototype for the function should be as follows:
int CALLBACK MyTickBoxFunction(TickBox *WhichTickBox);
The function will be passed a pointer to the tick box that was clicked and should return either
GUI_CONTINUE
or

GUI_END
.
flags: Currently, there are only three valid flags for tick boxes :- BG_SELECTED causes the tick box to be ticked initially. The BG_CLEAR flag causes the tick box to be clear (i.e. the same colour as the window). Clear tick boxes are drawn and refreshed more quickly than filled ones. The
S_AUTO_SIZE
flag causes the tick box to auto-size.
extension: This is reserved for future expansion and should be set to NULL.

Returns:

If successful, a pointer to the new tick box is returned. If the function fails then NULL is returned.

Known bugs:

None.

See also:

Destroy

SetPreText

SetPostText

TickBoxValue

1.90 FoxGUI Editbox functions

Edit boxes are Intuition string gadgets. They are containers into which the user can type text. If there are several in a window (and you are using OS version 2.0 or above) you can switch between edit boxes using tab and shift-tab. FoxGUI supplies three default filters for edit boxes: TEXT, INT and FLOAT. INT edit boxes are for capturing integral numbers. The user can type digits and a preceeding + or - sign only. All other characters are rejected. FLOAT also allows the . symobol allowing floating point numbers to be entered and TEXT allows anything to be typed. The default filter is TEXT and is supported under all versions of the Amiga OS. The INT filter is supported from release 2.00 onwards and the FLOAT filter is supported in Intuition version 36 and above. In a FLOAT filtered edit box, you can also specify the number of digits allowed after the decimal point using SetEditBoxDP. Functions are available for setting the text in edit boxes as well as for retrieving text or numbers that the user has entered.

The following editbox functions are currently available :-

GetEditBoxDouble

GetEditBoxID

GetEditBoxInt

GetEditBoxText

MakeEditBox

RefreshEditBox

SetEditBoxCols

SetEditBoxDP

SetEditBoxDouble

SetEditBoxFocus

SetEditBoxInt

SetEditBoxText
See also:

Destroy

DisableControl

EnableControl

1.91 GetEditBoxDouble function

Function prototype:

```
double GetEditBoxDouble(EditBox *p);
```

Description:

Converts the text in the specified edit box into a double-precision floating point number and returns the result. On Intuition version 36 or higher, an edit box created with type FLOAT_EDIT is filtered so that a user can only type floating point numbers into it. In this case GetEditBoxDouble will usually succeed (note that even under these circumstances it is possible for a FLOAT_EDIT edit box to contain text which cannot be interpreted as a floating point number but only if it has been set from within the program by use of the function

SetEditBoxText
. If the number cannot be interpreted as a floating point number then anything may be returned. You can control the number of decimal places that the user can type into a FLOAT_EDIT edit box using the function
SetEditBoxDP
.

Parameters:

p: A pointer to the edit box.

Returns:

A double-precision floating point number whose value is the text in the specified edit box.

Known bugs:

None.

See also:

GetEditBoxInt

GetEditBoxText

MakeEditBox

SetEditBoxDP

SetEditBoxDouble

1.92 GetEditBoxInt function

Function prototype:

```
int GetEditBoxInt(EditBox *p);
```

Description:

Converts the text in the specified edit box into an integer and returns the result. On OS version 2.00 or higher, an edit box created with type INT_EDIT is filtered so that a user can only type integral numbers into it. In this case GetEditBoxInt will usually succeed (note that even under these circumstances it is possible for an INT_EDIT edit box to contain text which cannot be interpreted as an integer but only if it has been set from within the program by use of the functions

SetEditBoxText
or
SetEditBoxDouble
. If the number cannot be interpreted as an integer then anything may be returned.

Parameters:

p: A pointer to the edit box.

Returns:

An integer whose value is the text in the specified edit box.

Known bugs:

None.

See also:

GetEditBoxDouble

GetEditBoxText

MakeEditBox

SetEditBoxInt

1.93 GetEditBoxText function

Function prototype:

```
char *GetEditBoxText(EditBox *p);
```

Description:

Get the current text in the specified edit box.

Parameters:

p: A pointer to the edit box.

Returns:

A pointer to a NULL terminated text string which is the text contained in the specified edit box. The pointer returned is a pointer to the actual buffer used by the edit box so you should never directly modify this string in any way. If you keep a copy of the pointer you should also remember that it will become invalid when the edit box is destroyed. The safest thing to do is make your own copy of the string using a function such as strcpy.

Known bugs:

None.

See also:

GetEditBoxDouble

GetEditBoxInt

MakeEditBox

SetEditBoxText

1.94 MakeEditBox function

Function prototype:

```
EditBox *MakeEditBox(void *Parent, int x, int y, int len, int buflen,
   int id, BOOL (*callfn) (EditBox*), long flags, void *extension);
```

Description:

Creates a new edit box in the specified window or frame.

Parameters:

Parent: A pointer to an open GuiWindow or frame in which to create the editbox.
x: The coordinate of the left edge of the new edit box relative to the left hand edge of the specified window/frame.
y: The coordinate of the top edge of the new edit box relative to the top edge of the specified window/frame. Note that this is from the very top of a window so a y coordinate of 0 would cause the editbox to be at least partly obscured by the window's title bar if it had one.
len: The length in pixels of the edit box. This length includes the border drawn around the edit box.
buflen: The maximum number of characters that the user will be able to type into the new edit box. This number cannot be more than 256.
id: This parameter can have any integral value. It won't affect the way the edit box behaves but it will get stored as part of the edit box structure and you can find out the value of any edit boxes id using the function
GetEditBoxID
. The main use for this is when creating arrays of edit boxes. For example, if you wanted the user to enter their address you might create an array of edit boxes like this:

```
EditBox *ebAddress[5];
int l;

for (l = 0; l < 5; l++)
{
   ebAddress[l] = MakeEditBox(MyWin, 80, 80 + (10 * l), 244, 30,
         l, AddrValidate, THREED | TEXT_EDIT, NULL);
}
```

Because l has been passed to MakeEditBox as the id for each address line, the five address lines will have different ids, ranging from 0 to 4. Now, let's say for example that you wanted to keep a record of the number of characters in each address line. You could define an array of integers like this:

```
int numAddrChars[5];
```

And use your validation function (see the callfn parameter below) for the address line edit boxes to update your array like this:

```
BOOL CALLBACK AddrValidate(EditBox *eb)
{
   // Find out which address line has been changed.
   int index = GetEditBoxID(eb);

   // Update the character count
```

```
   numAddrChars[index] = strlen(GetEditBoxText(eb));
   return TRUE;
}
```

Without the id parameter, you would have to check each edit box pointer of the array in turn against the pointer passed to the validation function which would be very innefficient.

callfn: A pointer to a validation function for the edit box. This function will be called whenever the edit box loses focus (e.g. if the user has been typing in this box and then presses the tab key to activate the next edit box or clicks elsewhere in the window using the mouse). The function should have the following prototype:

```
BOOL CALLBACK MyValidationFunction(EditBox *MyEditBox);
```

When FoxGUI activates your validation function it will pass a pointer to the edit box that triggered it so that you can use one validation function for more than one edit box (if you wish) and still determine which edit box has just been deactivated. In order to validate the data you will obviously need to know what the user has typed into your edit box and you can use one of the functions

GetEditBoxDouble
,
GetEditBoxInt
or

GetEditBoxText
to find this out. If you decide that what the user has typed is invalid, you may wish to tell the user so using the

GuiMessage
function and you may want to force the user to correct it by re-activating the edit box (see below). Of course, your validation function can perform other action apart from validation. If the edit box was of type INT_EDIT or FLOAT_EDIT you may want to use the number they have entered for some form of calculation and display the result in an output box or another edit box. The function can really do whatever you want it to. Unlike other call-back functions, this one should return TRUE or FALSE. As mentioned above, if the user enters invalid data you might want to re-activate the edit box to force them to correct it. If your function returns FALSE then FoxGUI will re-activate the edit box for you (overriding any calls made to the

SetEditBoxFocus
function). If the data is valid or you don't want the edit box re-activated for any other reason then you should return TRUE. You should take great care when returning FALSE from this function - if the user has deactivated the edit box by clicking on another control then that control will not get activated unless this function returns TRUE. For example, if the user types invalid data into an edit box and then clicks on a tick box, the tick box will not change value and it's call-back function will not be activated unless the

call-back function for the edit box returns TRUE. The only gadgets that the user will be able to activate under these conditions are gadgets in other applications, some system gadgets in the current application and scroll-bars. For example, the user could scroll a list box or resize the window while the edit box text was invalid but the focus would be returned to the edit box immediately afterwards. The function

SetEditBoxFocus

, when called from within an edit boxes call-back function is not as clever. If you want to set the focus back to the edit box that just lost it then return FALSE, don't use SetEditBoxFocus. If you want to set the focus to a different edit box then use SetEditBoxFocus but do it as near to the end of the function as possible (calling GuiMessage, for example after a call to SetEditBoxFocus would completely ruin the effect of the call to SetEditBoxFocus becuase the window popping up will cause the edit box to lose focus again). If you are going to do anything that causes the focus to go anywhere other than where the user is expecting it to go it is polite to tell the user why (with a call to GuiMessage for example) otherwise you could completely confuse your user.

flags: Currently, the following flags are available for edit boxes: THREED, TEXT_EDIT, INT_EDIT, FLOAT_EDIT, NO_EDIT,

S_AUTO_SIZE

and EB_CLEAR. Set the THREED flag if you want the border around the edit box to have a three dimensional look (it will appear slightly inset or pressed into the screen and will be drawn in the current FoxGUI pens which can be modified by calling

SetGuiPens

). If you do not select the THREED flag then you will get a simple rectangle drawn around the editable area in the current default border colour. EB_CLEAR specifies that the edit box will be clear (i.e. see-through). In other words, the background colour of the edit box will be the colour of the window or frame in which it was created. The other four flags are mutually exclusive. You can select at most one of the four but any of the four may be combined with the THREED and EB_CLEAR flags. All four _EDIT flags specify the filtering that will be applied to the edit box when the user types data into it. TEXT_EDIT allows the user to type absolutely any text into the edit box. INT_EDIT allows the user to enter integral numbers only. FLOAT_EDIT allows floating point or integral numbers to be entered (floating point numbers are those with a decimal point e.g. 3.14159). For FLOAT_EDIT edit boxes, the number of figures after the decimal point can be restricted using the function

SetEditBoxDP

. NO_EDIT prevents any text from being entered into the edit box at any time (whether the edit box is currently enabled or disabled) - I have no idea why you would want to use this. If you do not select any of the _EDIT flags then TEXT_EDIT is used by default.

extension: This is reserved for future expansion and should be set to NULL.

```
Returns:

   If successful a pointer to the new edit box is returned.  NULL is
   returned if MakeEditBox fails.

Known bugs:

   None.

See also:

                Destroy

                DisableControl

                EnableControl

                GetEditBoxDouble

                GetEditBoxInt

                GetEditBoxText

                RefreshEditBox

                SetEditBoxCols

                SetEditBoxDP

                SetEditBoxDouble

                SetEditBoxFocus

                SetEditBoxInt

                SetEditBoxText
```

1.95 RefreshEditBox function

```
               Function prototype:

   void RefreshEditBox(EditBox *p);

Description:

   Refresh the imagery of the specified edit box.  You might want to do
   this if you have used a function such as
                DisableM
                which has
   changed the state of one or more edit boxes without refreshing the
   imagery (all functions which change the state of an editbox can be
   instructed to refresh the imagery for you if you prefer).
```

```
Parameters:

   p: A pointer to the edit box to be refreshed.

Known bugs:

   None.

See also:

                Destroy

                DisableControl

                EnableControl
```

1.96 SetEditBoxCols function

```
               Function prototype:

   BOOL SetEditBoxCols(EditBox *p, int BorderCol, int Bcol, int Tcol);

Description:

   Changes the colours of an edit box to the new ones specified and
   refreshes the edit box imagery.

Parameters:

        p: A pointer to the edit box whose colours are to be changed.
BorderCol: The pen colour for the border.
     Bcol: The background pen colour for the edit box (ignored prior to
           Intuition version 37).
     Tcol: The pen colour for the text within the edit box.


Returns:

   TRUE for success, FALSE for failure.

Known bugs:

   None.

See also:

                MakeEditBox

                SetEditBoxDP

                SetEditBoxFocus
```

1.97 SetEditBoxDP function

```
                Function prototype:

   BOOL SetEditBoxDP(EditBox *p, int num);

Description:

   Set (or change) the maximum number of figures that can be entered after
   the decimal point in an edit box of type FLOAT_EDIT.  This function can
   be called at any point after the edit box is created so you should bear
   in mind that if the user has already had a chance to type text into the
   edit box then there may already be more digits after the decimal point
   than you want.  Calling this function will not remove any extra digits
   that are already after the decimal point but this can be done by using

                GetEditBoxDouble
                to get the current value in the edit box and then

                SetEditBoxDouble
                which will truncate the number to the correct number
   of decimal places when setting the new value.  For edit boxes of types
   other than FLOAT_EDIT, SetEditBoxDP won't prevent the user from typing
   more than the specified number of decimal places into the text box but
   it will affect the way
                SetEditBoxDouble
                behaves when it is called for
   that edit box.

Parameters:

     p: A pointer to the edit box to modify.
   num: The maximum number of digits to allow after the decimal point in
        the specified edit box.

Returns:

   TRUE for success, FALSE for failure.

Known bugs:

   None.

See also:


                GetEditBoxDouble

                MakeEditBox

                SetEditBoxDouble
```

1.98 SetEditBoxDouble function

```
                Function prototype:

   BOOL SetEditBoxDouble(EditBox *p, double num);

Description:

   Set the text in the specified edit box to the number supplied.  The
   number will be truncated to the maximum number of decimal places allowed
   in the edit box as set by the function
                SetEditBoxDP
               .  This is true for
   all edit boxes, not just those of type FLOAT_EDIT.  For example, the
   following code will cause the text in MyEditBox to be set to "3.14":

   float pi = 3.14159265;
   SetEditBoxDP(MyEditBox, 2);
   SetEditBoxDouble(MyEditBox, pi);

Parameters:

     p: A pointer to the edit box whose text is to be changed.
   num: The double-precision floating point number whose value is to be
        converted to text, possibly truncated and placed in the edit box.

Returns:

   TRUE for success, FALSE for failure.

Known bugs:

   None.

See also:


                GetEditBoxDouble

                MakeEditBox

                SetEditBoxDP

                SetEditBoxInt

                SetEditBoxText
```

1.99 SetEditBoxFocus function

```
                Function prototype:

   BOOL SetEditBoxFocus(EditBox *p);
```

```
Description:

   Attempts to activate the edit box specified.  If successful, a cursor
   will appear in the edit box and the user will then be able to type data
   into it.

Parameters:

   p: A pointer to the edit box to be activated.

Returns:

   SetEditBoxFocus will return TRUE if Intuition claims to have
   successfully activated the edit box.  Otherwise FALSE will be returned.
   Intuition might fail to activate the edit box if for example the user is
   holding the right mouse button down to display the menus.  FALSE will be
   returned if the edit box is currently disabled.

Known bugs:

   None.

See also:

   Notes on the use of SetEditBoxFocus in edit box call-back functions
   (
                MakeEditBox
               ).

                DisableControl

                EnableControl

                RefreshEditBox
```

1.100 SetEditBoxInt function

```
               Function prototype:

   BOOL SetEditBoxInt(EditBox *p, int num);

Description:

   Sets the edit box text to the number supplied.

Parameters:

     p: A pointer to the edit box whose value is to be changed.
   num: The number to convert to text and place in the edit box.

Returns:

   TRUE for success, FALSE for failure.
```

```
Known bugs:

   None.

See also:

                GetEditBoxInt

                MakeEditBox

                SetEditBoxDouble

                SetEditBoxText
```

1.101 SetEditBoxText function

```
               Function prototype:

   BOOL SetEditBoxText(EditBox *p, char *text);

Description:

   Set the text in the specified edit box to the string supplied.

Parameters:

      p: A pointer to the edit box whose value is to be changed.
   text: A pointer to a text string to copy into the edit box.  A copy
         will be made of the text string supplied so there is no need to
         preserve the string passed after calling the function.

Returns:

   TRUE for success, FALSE for failure.

Known bugs:

   None.

See also:

                GetEditBoxText

                MakeEditBox

                SetEditBoxDouble

                SetEditBoxInt
```

1.102 GetEditBoxID function

Function prototype:

```
int GetEditBoxID(EditBox *p)
```

Description:

Returns the id of the specified edit box. For a full description of edit box ids, see MakeEditBox.

Parameters:

p: A pointer to the edit box whose id you want to know.

Returns:

The id of the edit box specified.

See also:

MakeEditBox

1.103 FoxGUI Tree Control functions

Tree Controls are similar to ListBoxes in that they contain a list of items which can be selected with a single or double click and scrolled through with horizontal and vertical scrollers if the list contains more items than the box can show or if the items are too wide for the box. Unlike a list box, however, items are represented in a tree structure (rather like a family tree drawn sideways so that the parent items are on the left of the box and children are indented). Parent items in a tree control can be opened and closed so that their children are either visible or hidden - the user can do this by clicking on a small box with a plus or minus sign which will appear to the left of any item which has children. Clicking on the box will open or close the parent item so that the children are hidden or shown. The box will contain a - when the children are shown and a + when they are hidden. An image can also be shown to the left of the text of an item if you so wish. This is most commonly used when a tree control is used to represent a disk where the top level is the root directory of the disk (which might have a picture of a disk!) and the children are files and directories on that disk where directories would all have a common picture so that the user can tell at a glance that they are directories. Files might have different pictures which depend on the file type or may have no picture at all. In a tree control, user defined data can be associated with any item in the control. For example, going back to our disk example, the programmer may have gathered information about the size of each file and the file attributes. He might choose to store these

in a structure for which he has one instance per file and the data
associated with the item in the tree control could be a pointer to the
relevant instance of that structure. In this way it would be wasy for the
programmer to show the user the size and attributes of the file selected.

The following tree control functions are currently available :-

AddItem

ClearTreeControl

CloseItem

FindTreeItem

ItemData

ItemIsOpen

MakeTreeControl

OpenItem

RemoveItem

ReplaceTCItem

SetTreeControlDragPointer

SetTreeControlHiItem

TCHiItem

TCHiText

TCItemText

See also:

Destroy

Hide

Show

DisableControl

EnableControl

1.104 AddItem function

Function prototype:

```
   TreeItem *AddItem(TreeControl *tc, TreeItem *InsBefore, TreeItem
      *Parent, char *text, BOOL IsOpen, GuiBitMap *bm, void *ItemData);
```

Description:

Adds an item to a tree control.

Parameters:

```
          tc: A pointer to the tree control.
   InsBefore: If ordering of items is important in your tree control, you
              can pass a pointer to an existing item in the tree control
              and the new item will be inserted before it.  If ordering is
              not important, set this to NULL.
      Parent: If the new item is to be a child of an existing item, pass a
              pointer to that item, otherwise NULL.
        text: The text of the item to add.
      IsOpen: Whether or not the item is open.  This has no effect when you
              first add the item but if you subsequently add an item as a
              child of this one, then this item will default to being
              closed (i.e. not showing the child) unless you set this to
              TRUE.
          bm: A pointer to a bitmap to display to the left of the item's
              text.  If the bitmap is taller than the font used for the
              tree control then the bitmap will be scaled down to the
              height of the text and the width will also be scaled
              proportionately.  You should load the bitmap using the

                 LoadBitMap
                 function and you should not free the bitmap
              (using the
                 FreeGuiBitMap
                 function) until either the item has
              been removed from the tree control or the tree control has
              been destroyed.
    ItemData: A pointer to data associated with this item.  You can
              retrieve a pointer to the item data of the hilighted item
              by calling ItemData(TCHiItem(MyTreeControl)).  Pass NULL if
              you do not want to associate any data with this item.
```

Returns:

If successful, a pointer to the new tree item, otherwise NULL.

Known bugs:

None.

See also:

ClearTreeControl

CloseItem

ItemData

```
ItemIsOpen

MakeTreeControl

OpenItem

RemoveItem

SetTreeControlHiItem

TCHiItem

TCHiText

TCItemText
```

1.105 ClearTreeControl function

```
Function prototype:

   void ClearTreeControl(TreeControl *tc);

Description:

   Clears a tree control of all items.

Parameters:

   tc: A pointer to the tree control to clear.

Known bugs:

   None.

See also:

Destroy

RemoveItem
```

1.106 CloseItem function

```
Function prototype:

void CloseItem(TreeItem *it);

Description:
```

```
   Close an open tree control item, hiding it's children.  This function
   has exactly the same effect as the user clicking on the +/- button of
   an open tree control item.

Parameters:

   ti: A pointer to the item to close.

Known bugs:

   None.

See also:

                OpenItem
```

1.107 FindTreeItem function

```
               Function prototype:

   TreeItem *FindTreeItem(TreeControl *tc, char *text);

Description:

   Finds a tree item which has the text specified.

Parameters:

     tc: A pointer to the tree control.
   text: The text to search for.

Returns:

   If successful, a pointer to the relevant tree item is returned.
   Otherwise NULL.

Known bugs:

   None.

See also:

                AddItem

                TCItemText
```

1.108 ItemData function

Function prototype:

```
void *ItemData(TreeItem *ti);
```

Description:

Returns the item data associated with the tree item. This will be the pointer which was passed as the ItemData parameter to the AddItem function.

Parameters:

ti: A pointer to the tree item.

Returns:

A pointer to the item data or NULL if none was supplied.

Known bugs:

None.

See also:

AddItem

TCHiItem

1.109 ItemIsOpen function

Function prototype:

```
BOOL ItemIsOpen(TreeItem *it);
```

Description:

Returns TRUE if the children of this item are theoretically visible. I.e. if this item has a parent then the parent must also be open and if that item has a parent then it must also be open etc. If the item has no children but it's parent (if it has one) is open and so is it's parent etc all the way to the top of the tree then this function will return the value of IsOpen that was passed to the AddItem function when this item was created.

Parameters:

ti: A pointer to the item.

Returns:

TRUE or FALSE (see above).

```
Known bugs:

   None.

See also:

                AddItem

                CloseItem

                OpenItem
```

1.110 MakeTreeControl function

```
               Function prototype:

   TreeControl *MakeTreeControl(void *Parent, int left, int top, int width, int ↩
      height,
      int lborder, int tborder, int flags, int (*Eventfn) (TreeControl*,
      short, TreeItem*, void**), void *extension);

Description:

   Make a new FoxGUI tree control.

Parameters:

     Parent: A pointer to an open FoxGUI window or frame in which to
             create the new tree control.
       left: The x coordinate of the left edge of the tree control
             relative to the left edge of the window/frame.
        top: The y coordinate of the top edge of the tree control relative
             to the top edge of the window/frame.
      width: The width of the tree control in pixels.  Note that this
             includes the width of the scroll gadget on the right hand edge
             when there is one.
     height: The height of the tree control in pixels.
    lborder: Left border - the distance in pixels between the left border
             of the tree control and the left edge of the text for the
             items within it.  2 or 3 pixels is usually sufficient to make
             it look neat.
    tborder: Top border - the distance in pixels between the top border of
             the tree control and the top edge of the text for the first
             item in the list box.  1 or 2 pixels is usually sufficient to
             make it look neat.
      flags: The following flags are currently available for tree controls:
             TC_SELECT, TC_CURSOR, TC_DBLCLICK, TC_DRAG, TC_DROP,
             TC_OPENITEM, TC_CLOSEITEM,
                 S_AUTO_SIZE
                 and
             TC_REHILIGHT_ON_SCROLL.

             The TC_SELECT flag should be specified if you want a function
```

of your own to be called whenever a user clicks on an item in the tree control. The TC_CURSOR flag should be specified if you want your function to be called when the user changes the currently hilighted item without clicking on an item (i.e. by using the cursor keys or dragging a scroll bar). In the case of dragging a scroll bar, if a new item is hilighted, your function will be called once when the user releases the scroll bar. The TC_DRAG flag should be specified if you want the user to be able to drag data out of this tree control into other drag/drop aware controls. If you wish to supply your own mouse pointer for use during drag/drop actions then see

SetTreeControlDragPointer

. If you want to be able to drop data dragged from other drag/drop aware controls into this tree control then specify the TC_DROP flag. If you want to specify an action to occur when a user double clicks on an item in the tree control then specify the TC_DBLCLICK flag. If you want to know when a user clicks on an open or close (+/-) button on an item in the tree control then use TC_OPENITEM and TC_CLOSEITEM. These two flags will cause your event function to be called when the user clicks on a +/- button, before FoxGUI has opened the item. The TC_REHILIGHT_ON_SCROLL flag ensures that the hilighted item in a tree control is always within the visible part of the list so that if scrolling the list would cause the hilighted item to be outside the visible portion of the list then a new item will be hilighted instead. If you specify any of these flags (other than TC_REHILIGHT_ON_SCROLL) then you will need to specify and EventFn (see below).

Eventfn: A pointer to a function to handle all tree control events. This parameter should be specified if any of the following flags have been specified: TC_SELECT, TC_CURSOR, TC_DBLCLICK, TC_DRAG, TC_DROP, TC_OPENITEM or TC_CLOSEITEM. The function should have the following prototype:

```
int CALLBACK Eventfn(TreeControl *tc, short Event,
   TreeItem *HiItem, void **data);
```

and will be called whenever one of those events occurs and will be passed a pointer to the tree control (so that your function can be used to handle events for more than one tree control). The Event parameter will be one of TC_SELECT, TC_CURSOR, TC_DBLCLICK, TC_DRAG, TC_DROP, TC_OPENITEM or TC_CLOSEITEM depending on which event occured and HiItem will contain a pointer to the item that was selected, double clicked on, opened, closed or dragged out of the box or, in the case of the TC_DROP event, a pointer to the item above which the cursor was positioned when the user let go of the mouse button. In the case of the TC_DROP event, HiItem will be NULL if the drop occurred in an area where there was no item. If data has been dragged from another drag/drop aware control and dropped in this one then *data will be a pointer to the data that was dragged which would have been set in the event function for the originating control. If data is being dragged from this control into another drag/drop aware control (the TC_DRAG event) then you can set *data to point to anything you want

```
            and if the drop event occurs above another drag/drop aware
            control, that pointer will be passed to the event function of
            that control.
            As with almost all user-defined functions called directly by
            FoxGUI, this function should return either
                 GUI_CONTINUE
                            or
                 GUI_END
                 but you should see the notes in the

                 Drag/Drop functionality
                 section about return values from
            drag event functions.
 extension: This is reserved for future expansion and should be set to
            NULL.

Returns:

   If successful, a pointer to a new FoxGUI tree control.  NULL otherwise.

Known bugs:

   None.

See also:


                 Drag/Drop functionality

                 AddItem

                 RemoveItem

                 TCHiItem

                 TCHiText

                 SetTreeControlDragPointer

                 SetTreeControlHiItem

                 Destroy

                 Hide

                 Show

                 DisableControl

                 EnableControl
```

1.111 SetTreeControlDragPointer function

```
                Function prototype:

   void SetTreeControlDragPointer(TreeControl *tc, unsigned short
      *DragPointer, int width, int height, int xoffset, int yoffset);

Description:

   Specify a custom mouse-pointer to be shown when dragging data out of a
   drag/drop enabled tree control.

Parameters:

            tc: A pointer to the tree control.
   DragPointer: A pointer to an array of numbers making up a standard
                Intuition sprite data structure.  This must be stored in
                chip memory since it is to be used as a mouse pointer.
         width: The width in pixels of the pointer provided.  The maximum
                width of an Amiga mouse pointer is 16 pixels.
        height: The height in pixels of the pointer provided.  There is no
                maximum height.
   int XOffset:
   int YOffset: These two numbers specify the offset of the pointers
                hot-spot from the top left corner of the sprite.  They are
                typically zero or negative.

Known bugs:

   None.

See also:


                 Drag/Drop functionality

                 MakeTreeControl
```

1.112 OpenItem function

```
                Function prototype:

   void OpenItem(TreeItem *it);

Description:

   Open a closed tree control item, show it's children.  This function
   has exactly the same effect as the user clicking on the +/- button of
   an closed tree control item.

Parameters:

   ti: A pointer to the item to open.

Known bugs:
```

None.

See also:

CloseItem

1.113 RemoveItem function

Function prototype:

```
void RemoveItem(TreeItem *ti);
```

Description:

Remove the specified item from the tree control.

Parameters:

ti: The item to remove.

Known bugs:

None.

See also:

AddItem

1.114 ReplaceTCItem function

Function prototype:

```
TreeItem *ReplaceTCItem(TreeItem *old, char *text, GuiBitMap *bm, void * ↩
    ItemData);
```

Description:

Replaces an item from a tree control with a new item described by the text, bm and ItemData parameters.

Parameters:

old: A pointer to the tree item to replace.

All of the other parameters are equivalent to the ones for the AddItem function.

```
Returns:

   A pointer to the new item or NULL if not successful.  Note that after
   calling this function the pointer to the old item is no longer valid and
   should not be used.

   From version 4.7 onwards, it is safe to pass the return value from

                TCItemText
                for this item as the text parameter of this function.  This
   allows you to easily change the image of an item without changing the
   text.  For example:
   newitem = ReplaceTCItem(olditem, TCItemText(olditem), newbitmap, NULL);

Known bugs:

   None.

See also:


                AddItem

                FindTreeItem

                RemoveItem
```

1.115 SetTreeControlHiItem function

```
               Function prototype:

   void SetTreeControlHiItem(TreeControl *tc, TreeItem *HiItem, BOOL refresh);

Description:

   Change or set the currently hilighted item of a tree control.

Parameters:

        tc: A pointer to the tree control.
    HiItem: A pointer to the tree item to hilight.
   refresh: If TRUE, refresh the tree control to show the newly hilighted
            item, otherwise it will be shown next time the tree control is
            refreshed.

Known bugs:

   None.

See also:


                TCHiItem
```

TCHiText

1.116 TCHiItem function

Function prototype:

```
TreeItem *TCHiItem(TreeControl *tc);
```

Description:

Return a pointer to the currently hilighted item of a tree control.

Parameters:

tc: A pointer to the tree control.

Returns:

A pointer to the hilighted item or NULL if no item is hilighted.

Known bugs:

None.

See also:

SetTreeControlHiItem

TCHiText

1.117 TCHiText function

Function prototype:

```
char *TCHiText(TreeControl *tc);
```

Description:

Return the text of the currently hilighted item in a given tree control.
Note that:

```
char *hitext = TCHiText(MyTreeControl);
```

is exactly equivalent to:

```
char *hitext = TCItemText(TCHiItem(MyTreeControl));
```

Parameters:

tc: A tree control.

Returns:

```
   A pointer to the text string of the currently hilighted item or NULL if
   no item is hilighted.

Known bugs:

   None.

See also:

                SetTreeControlHiItem

                TCHiItem

                TCItemText
```

1.118 TCItemText function

```
               Function prototype:

   char *TCItemText(TreeItem *ti);

Description:

   Return the text of a specified tree item.

Parameters:

   ti: The tree item.

Returns:

   A pointer to the text string of the tree control item.

Known bugs:

   None.

See also:

                TCHiItem

                TCHiText
```

1.119 FoxGUI Listbox functions

```
               List boxes consist of a frame around a list of items.  When a user ↩
                   clicks
```

on an item in the list it becomes hilighted and (optionally) a function can be triggered. A function can also be triggered by a double-click if required. If there are more items in a list than can be shown in the frame, then the list box will automatically get a scroll bar (proportional gadget) on it's right hand edge which can be used to scroll up and down the list of items. If an item is added to the list box which is wider than the box itself, the list box will get a horizontal scroll bar (proportional gadget) on the bottom edge.

Functions are supplied to sort the items in a list box (into numerical or alphabetical, ascending or descending order) and it is possible to have a list box arranged in columns by setting tab-stops.

List boxes are drag-drop aware. Items can be dragged into or out of list boxes (see
Drag/Drop functionality
).

The following listbox functions are currently available :-

AddListBoxItem

AddListBoxTitle

ClearListBoxItems

ClearListBoxTabStops

ClearListBoxTitles

FindListText

HiElem

HiNum

HiText

InsertListBoxItem

ListBoxRefresh

ListColumnText

MakeListBox

NoLines

NoTitles

ReplaceListBoxItem

SetListBoxHiElem

SetListBoxHiNum

SetListBoxTabStopsArray

SetListBoxTopNum

SortListBox

TopNum
See also:

Destroy

DisableControl

EnableControl

1.120 AddListBoxItem function

Function prototype:

```
ListBoxItem *AddListBoxItem(ListBox *nlb, char *item, BOOL
   refresh);
```

Description:

Adds a line of text to an existing FoxGUI listbox. The item is added to the end of the list but the list can be sorted if you require it once all of the required items have been added. The items in a list box can contain tabs to align data in columns. Items in a list box are sometimes referred to in this manual as "elements".

Parameters:

nlb: The FoxGUI list box to which to add the item.

item: The text string to add to the list box. If the text string contains tab characters (specified as '\t' in C) then each tab will cause the character following it to be printed at the next tab stop as specified when calling the function

SetListBoxTabStopsArray
. For a full example of using tabbed lists see the
SetListBoxTabStopsArray
function (which should be called before any elements are added). If you add more items than can fit within the list box then a scroll bar is automatically created at the right hand edge of the list box.

refresh: If TRUE then the list box will be refreshed after this item has been added so that you see the item immediately (or you see the scrollbar resize if the item is outside the currently visible portion of the list box). If you are adding many items to a list box at once then it is quicker to add them all without refreshing and then do one refresh at the end - either by setting refresh to TRUE for the very last item added or by

```
            calling the function
                 ListBoxRefresh
                 after adding the last
            item.

Returns:

   If successful, a pointer to the item which has been added, otherwise
   NULL.  There is no good reason for maintaining a pointer to each item
   that you add because the listbox will do that for you but it may be
   important for your application to check this function for a non-NULL
   result just to check that it has succeeded.

Known bugs:

   None.

See also:


                 AddListBoxTitle

                 ClearListBoxItems

                 HiElem

                 HiNum

                 HiText

                 InsertListBoxItem

                 ListBoxRefresh

                 MakeListBox

                 NoLines

                 SetListBoxHiNum

                 SetListBoxTabStopsArray

                 SetListBoxTopNum

                 SortListBox

                 TopNum
```

1.121 AddListBoxTitle function

```
               Function prototype:

   BOOL AddListBoxTitle(ListBox *nlb, char *title, BOOL refresh);
```

Description:

Adds a title line to the specified list box. Unlike the items added to a list box, title lines cannot be sorted (they are always shown at the top of the list in the order that they were added) and are not scrolled by the scroll-bar (if present) on the right hand edge of the list box. When a list box is created, the Gui will calculate the maximum number of lines of text that can be shown in the list at a time (which will depend on the height of the list and the font size specified for the text) and will not allow the titles to fill the visible list space. In other words the Gui always allows room for at least one item to be shown in the list at a time. Any attempt to add a title to the last visible line of a list box will fail. In practice, title lines are usually far outnumbered by visible items if they are present at all. It is not necessary to have any titles on a list box if you prefer. As with list box items, if the list box has tab stops and the title specified has tab characters in it then the character immediately following each tab character will appear at the next tab stop set. In this way it is possible to have columns of text or data aligned with titles at the top.

Parameters:

nlb: A pointer to the FoxGUI list box to which to add the title.
title: The text string to add as a title to the specified list box.
refresh: If TRUE then the list box will be refreshed after adding the title so that you see it immediately. If you want to add more than one title or a combination of titles and items at the same time then it is quicker not to refresh the list box after each but to wait until after adding the last one and then refresh the list box either by setting refresh to TRUE for the last title/item added or by calling
ListBoxRefresh
after
adding it.

Returns:

TRUE for success, FALSE for failure.

Known bugs:

None.

See also:

AddListBoxItem

ClearListBoxTitles

ListBoxRefresh

MakeListBox

NoTitles

SetListBoxTabStopsArray

1.122 ClearListBoxItems function

```
                Function prototype:

   void ClearListBoxItems(ListBox *lb, BOOL refresh);

Description:

   Removes all items from a FoxGUI list box and frees all associated
   resources.  This function does not clear the list boxes titles if any
   have been set.

Parameters:

        lb: A pointer to the list box whose items are to be removed.
   refresh: If TRUE then the list box will be refreshed so that the user
            will see an empty list box.  If you are emptying it to refill
            it again with different data then you may prefer to set refresh
            to FALSE and then refresh the list box after adding the new
            data.

Known bugs:

   None.

See also:


                AddListBoxItem

                ClearListBoxTabStops

                ClearListBoxTitles

                ListBoxRefresh

                MakeListBox

                NoLines

                NoTitles
```

1.123 ClearListBoxTabStops function

```
                Function prototype:

   void ClearListBoxTabStops(ListBox *nlb, BOOL refresh);

Description:
```

```
   Clears any tab stops previously set for a list box.

Parameters:

       nlb: A pointer to the list box whose tab stops are to be cleared.
   refresh: If TRUE the list box will be redrawn.  The text for items in a
            list box is formatted at the time that the items are added to
            the list box - to do this every time the list box redraws is
            too slow.  As a result, clearing the tab stops and refreshing
            the list will cause the existing items and titles in the list
            to be displayed exactly as though the tab stops still existed.
            Only titles and items added to the list box after calling this
            function will be affected.  It is therefore unlikely that you
            will need to set refresh to TRUE when clearing the tab stops
            unless you have also made other changes to the list box which
            you haven't yet refreshed.

Known bugs:

   None.

See also:

                AddListBoxItem

                AddListBoxTitle

                ClearListBoxItems

                ClearListBoxTitles

                ListBoxRefresh

                MakeListBox

                SetListBoxTabStopsArray
```

1.124 ClearListBoxTitles function

```
              Function prototype:

   void ClearListBoxTitles(ListBox *lb, BOOL refresh);

Description:

   Removes all titles from a FoxGUI list box and frees all associated
   resources.  This function does not clear any items from the list box.

Parameters:

        lb: A pointer to the list box whose titles are to be removed.
   refresh: If TRUE then the list box will be refreshed so that the user
```

will see the list box without it's titles immediately. If you are removing them to replace them with alternative titles then you may prefer to set refresh to FALSE and then refresh the list box after adding the new titles.

Known bugs:

None.

See also:

AddListBoxTitle

ClearListBoxItems

ClearListBoxTabStops

ListBoxRefresh

MakeListBox

NoTitles

1.125 FindListText function

Function prototype:

```
int FindListText(ListBox *lb, char *text, int reqcolumn);
```

Description:

Find the item number of the list box item which matches the specified text string.

Parameters:

lb: A pointer to the list box.
text: A pointer to the text item to be found. If the list box is in columns, a match will be found if the text matches the entire contents of a column. If the list is not in columns then a match will be found if the text matches the entire text for a line of the list box.
reqcolumn: If you want to find text in a specific column of a list box with tabs then set this to the column number required. If a match is found in a column other than the one specified then it won't be returned. If you want to find text in any column, set this parameter to 0. The left most column is column number 1.

Returns:

The item number of the list box item which matches the specified text. List box item numbers start at 1. If the item is not found, 0 is

```
   returned.

Known bugs:

   None.

See also:

                 ListColumnText
```

1.126 HiElem function

```
               Function prototype:

   ListBoxItem *HiElem(ListBox *lb);

Description:

   This function returns a pointer to the currently hilighted item in the
   specified FoxGUI list box.  If there is no hilighted item or the
   function fails for any other reason then NULL is returned.  This
   function is unlikely to be useful for your FoxGUI applications.  The
   functions
                HiNum
                and
                HiText
                are likely to be more useful.

Parameters:

   lb: A pointer to a FoxGUI list box.

Returns:

  A pointer to the currently hilighted element.  Note that if the list box
  has tab stops set and the hilighted element contains tabs then the line
  of the list box will consist of more than one element and the pointer
  returned is the pointer to the first of these.

Known bugs:

   None.

See also:

                 AddListBoxItem

                 ClearListBoxItems

                 ClearListBoxTabStops

                 HiNum
```

```
                HiText

                MakeListBox

                SetListBoxHiNum

                SetListBoxTabStopsArray

                TopNum
```

1.127 HiNum function

```
               Function prototype:

   int HiNum(ListBox *lb);

Description:

   Returns the number of the currently hilighted item in a FoxGUI list box.
   If there is no currently hilighted item or if any other error occurs, 0
   is returned (list box items are numbered starting at 1 not 0).

Parameters:

   lb: A pointer to a FoxGUI list box.

Returns:

   The number of the hilighted item.

Known bugs:

   None.

See also:

                AddListBoxItem

                ClearListBoxItems

                HiElem

                HiText

                MakeListBox

                NoLines

                NoTitles

                SetListBoxHiNum
```

SetListBoxTopNum

TopNum

1.128 HiText function

Function prototype:

```
char *HiText(ListBox *lb);
```

Description:

Returns a pointer to a text string containing the text of the currently hilighted item in the list box.

Parameters:

lb: A pointer to a FoxGUI list box.

Returns:

The text of the hilighted item in the list box. Note that this is a pointer to the text actually used by the list box itself so if you want to compare this or output it directly in some way then that's fine but if you need to modify it at all then you should take a copy using strcpy() or similar. Note also that if the list box has tab stops and the hilighted entry contains tabs then the text returned is only the text for the first column (i.e. up to the first tab stop). You can retrieve the text in other columns using the function

ListColumnText
.

Known bugs:

None.

See also:

HiElem

HiNum

ListColumnText

MakeListBox

SetListBoxHiNum

1.129 InsertListBoxItem function

Function prototype:

```
ListBoxItem *InsertListBoxItem(ListBox *nlb, char *item, ListBoxItem *after,  ↩
    BOOL refresh);
```

Description:

Inserts an item into a listbox, at a point specified by passing a pointer to the previous item.

Parameters:

```
     nlb: The listbox in which to insert the new item.
    item: The text of the new item.
   after: A pointer to the item after which to insert the new item.  If
          this parameter is NULL then the new item will be inserted at
          the beginning of the list.
 refresh: If TRUE, will refresh the listbox to show the new item.
```

Returns:

If successful, a pointer to the new item. NULL otherwise.

Known bugs:

None.

See also:

AddListBoxItem

ListBoxRefresh

ReplaceListBoxItem

1.130 ListBoxRefresh function

Function prototype:

```
void ListBoxRefresh(ListBox *lb);
```

Description:

Refreshes a list box. Most list box functions that change a list box in any way take a boolean parameter that specifies whether or not to refresh the list box afterwards. This is so that you can make multiple changes (e.g. add loads of items) without refreshing and then just refresh once at the end (which is faster than doing a refresh for each change).

```
Parameters:

   lb:  A pointer to the FoxGUI list box to refresh.

Known bugs:

   None.

See also:

                AddListBoxItem

                AddListBoxTitle

                ClearListBoxItems

                ClearListBoxTabStops

                ClearListBoxTitles

                MakeListBox

                SetListBoxHiNum

                SetListBoxTabStopsArray

                SetListBoxTopNum

                SortListBox
```

1.131 ListColumnText function

```
               Function prototype:

   char *ListColumnText(ListBox *lb, int col);

Description:

   Returns the text of the specified column of the hilighted item in the
   specified list box.

Parameters:

    lb: A pointer to the list box.
   col: The column number of the column whose text is to be returned.  The
        left most column is column number 0.

Returns:

   A pointer to the text of the specified column of the hilighted item.
   This is a pointer to the actual text used by the list box so if you
   intend to modify it you should make a copy of it first and modify the
   copy.
```

```
Known bugs:

   None.

See also:

                FindListText

                HiText

                SetListBoxHiElem

                SetListBoxHiNum

                SetListBoxTabStopsArray
```

1.132 MakeListBox function

```
               Function prototype:

   ListBox *MakeListBox(void *Parent, int left, int top, int width, int height,  ↩
      int lborder,
      int tborder, int flags, int (*Eventfn) (ListBox*, short, int, void**),
      void *extension);

Description:

   Make a new FoxGUI list box.

Parameters:

     Parent: A pointer to an open FoxGUI window or frame in which to
             create the new list box.
       left: The x coordinate of the left edge of the list box relative to
             the left edge of the window/frame.
        top: The y coordinate of the top edge of the list box relative to
             the top edge of the window/frame.
      width: The width of the list box in pixels.  Note that this includes
             the width of the scroll gadget on the right hand edge when
             there is one.
     height: The height of the list box in pixels.
    lborder: Left border - the distance in pixels between the left border
             of the list box and the left edge of the text for the titles
             and items within it.  2 or 3 pixels is usually sufficient to
             make it look neat.
    tborder: Top border - the distance in pixels between the top border of
             the list box and the top edge of the text for the first
             title/item in the list box.  1 or 2 pixels is usually
             sufficient to make it look neat.
      flags: The following flags are currently available for list boxes:
             LB_SELECT, LB_CURSOR, LB_DBLCLICK, LB_DRAG, LB_DROP,
             LB_REHILIGHT_ON_SCROLL and
```

S_AUTO_SIZE
.
The LB_SELECT flag should be specified if you want a function of your own to be called whenever a user clicks on an item in the list box. The LB_CURSOR flag should be specified if you want your function to be called when the user changes the currently hilighted item without clicking on an item (i.e. by using the cursor keys or dragging a scroll bar). In the case of dragging a scroll bar, if a new item is hilighted, your function will be called once when the user releases the scroll bar. The LB_DRAG flag should be specified if you want the user to be able to drag data out of this list box into other drag/drop aware controls. If you wish to supply your own mouse pointer for use during drag/drop actions then call the function
SetListBoxDragPointer
. If you want to be able to drop data dragged from other drag/drop aware controls into this list box then specify the LB_DROP flag. If you want to specify an action to occur when a user double clicks on an item in the list box then specify the LB_DBLCLICK flag. The LB_REHILIGHT_ON_SCROLL flag ensures that the hilighted item in a list box is always within the visible part of the list so that if scrolling the list would cause the hilighted item to be outside the visible portion of the list then a new item will be hilighted instead.

Eventfn: A pointer to a function to handle all list box events. This parameter should be specified if any of the following flags have been specified: LB_SELECT, LB_CURSOR, LB_DBLCLICK, LB_DRAG or LB_DROP. The function will be called whenever one of those events occurs and will be passed a pointer to the list box (so that your function can be used to handle events for more than one list box). The function should have the following prototype:

```
int CALLBACK EventFn(ListBox *lb, short Event, int
   ItemNum, void **data);
```

The Event parameter will be one of LB_SELECT, LB_CURSOR, LB_DBLCLICK, LB_DRAG or LB_DROP depending on which event occured and ItemNum will contain the item number of the item that was selected, double clicked on or dragged out of the box or, in the case of the LB_DROP event, the item number of the item above which the cursor was positioned when the user let go of the mouse button. In the case of the LB_DROP event, ItemNum will be zero if the drop occurred in an area where there was no item (for example if the drop occurred over one of the list boxes titles). If data has been dragged from another drag/drop aware control and dropped in this one then *data will be a pointer to the data that was dragged which would have been set in the event function for the originating control. If data is being dragged from this control into another drag/drop aware control (the LB_DRAG event) then you can set *data to point to anything you want and if the drop event occurs above another drag/drop aware control, that pointer will be passed to the event function of that control. As with almost all user-defined functions called directly by

```
              FoxGUI, this function should return either
                  GUI_CONTINUE
                                or
                  GUI_END
                  but you should see the notes in the

                  Drag/Drop functionality
                  section about return values from drag
              event functions.
  extension: This is reserved for future expansion and should be set to
              NULL.

Returns:

   If successful, a pointer to a new FoxGUI list box.  NULL otherwise.

Notes:

   Prior to release 5.0, it was possible to pass extra parameters to define
   a mouse pointer which would be shown when dragging data out of the list
   box.  This can now be achieved by calling the function

                  SetListBoxDragPointer
                 .

Known bugs:

   None.

See also:


                  Drag/Drop functionality

                  AddListBoxItem

                  AddListBoxTitle

                  ClearListBoxItems

                  ClearListBoxTabStops

                  ClearListBoxTitles

                  Destroy

                  DisableControl

                  EnableControl

                  HiElem

                  HiNum

                  HiText

                  ListBoxRefresh
```

NoLines

NoTitles

SetListBoxDragPointer

SetListBoxHiNum

SetListBoxTabStopsArray

SetListBoxTopNum

SortListBox

TopNum

1.133 SetListBoxDragPointer function

Function prototype:

```
   void SetListBoxDragPointer(ListBox *lb, unsigned short *DragPointer,
      int width, int height, int xoffset, int yoffset);
```

Description:

Sets the mouse-pointer to be shown when dragging data out of the specified list box if it is drag/drop enabled.

Parameters:

```
   lb:          A pointer to a FoxGUI list box.
   DragPointer: A pointer to an array of numbers making up a standard
                Intuition sprite data structure.  This must be stored in
                chip memory since it is to be used as a mouse pointer.
         width: The width in pixels of the pointer provided.  The maximum
                width of an Amiga mouse pointer is 16 pixels.
        height: The height in pixels of the pointer provided.  There is no
                maximum height.
       xoffset:
       yoffset: These two numbers specify the offset of the pointers
                hot-spot from the top left corner of the sprite.  They are
                typically zero or negative.
```

Known bugs:

None.

See also:

MakeListBox

1.134 NoLines function

Function prototype:

```
int NoLines(ListBox *lb);
```

Description:

Uses the list boxes height, top border, font size and number of titles to calculate the number of items that can be shown in the visible area of a list box. (Obviously this does not determine the maximum number of items that can be added to a list box because the visible portion of a list box can be scrolled).

Parameters:

lb: A pointer to a FoxGUI list box.

Returns:

The number of items that can be displayed in the list box.

Known bugs:

None.

See also:

MakeListBox

NoTitles

1.135 NoTitles function

Function prototype:

```
int NoTitles(ListBox *lb);
```

Description:

Returns the number of titles currently displayed in the specified list box.

Parameters:

lb: A pointer to a FoxGUI list box.

Returns:

The number of titles currently displayed in the specified list box.

Known bugs:

None.

See also:

AddListBoxTitle

ClearListBoxTitles

MakeListBox

NoLines

1.136 ReplaceListBoxItem function

Function prototype:

```
ListBoxItem *ReplaceListBoxItem(ListBox *nlb, char *item, ListBoxItem
   *OldItem, BOOL refresh);
```

Description:

Replaces a list box item. The specified item is replaced by the new item text specified.

Parameters:

```
    nlb: A pointer to the list box.
   item: A pointer to the text of the new item.
OldItem: A pointer to the item to replace.
refresh: TRUE if you want the list box to be redrawn to show the
         replacement item, FALSE otherwise.
```

Returns:

A pointer to the new list box item. Once an item has been replaced, the pointer to it becomes meaningless and should not be used. The pointer returned by this function should be used in it's place.

Known bugs:

None.

See also:

AddListBoxItem

ClearListBoxItems

InsertListBoxItem

FindListText

HiElem

ListBoxRefresh

1.137 SetListBoxHiElem function

Function prototype:

```
void SetListBoxHiElem(ListBox *lb, ListBoxItem *item, BOOL refresh);
```

Description:

Sets the hilighted item of the list box to the element specified.

Parameters:

```
    lb: A pointer to the list box.
  item: A pointer to the element to be hilighted.
refresh: TRUE if you want the list box to be redrawn to show the new
         hilighted element, FALSE otherwise.
```

Known bugs:

None.

See also:

AddListBoxItem

HiElem

HiNum

HiText

ListBoxRefresh

SetListBoxHiNum

SetListBoxTopNum

1.138 SetListBoxHiNum function

Function prototype:

```
void SetListBoxHiNum(ListBox *lb, int num, BOOL refresh);
```

Description:

Hilight the specified item number in the specified list box. This function will fail if you specify a number greater than the number of items in the list box. Note that if the item number specified is not currently in the visible portion of the list box, the list box will not be automatically scrolled to show the hilighted item. You can do this yourself using the function

SetListBoxTopNum

.

You can use a combination of other list box functions to work out whether or not your target item number is currently in the visible portion of the list box as follows :-

```
if (itemnum >= TopNum(lb) && itemnum <= NoLines(lb) + TopNum(lb) - 1)
{
   // itemnum is in the visible portion of list box lb.
}
```

List box items start at 1. If you pass num as 0, the hilighted item (if any) will be unhilighted but no new item will be hilighted.

Parameters:

lb: A pointer to a FoxGUI list box.
num: The item number of the item to hilight.
refresh: If TRUE, refresh the list box to unhilight the previously selected item and hilight the selected item. Otherwise, leave the list box looking as it was. Typically you would set this to FALSE if you had many other changes to make to the list box - it's quicker to make all of the changes without refreshing and then refresh the list box just once at the end.

Known bugs:

None.

See also:

AddListBoxItem

HiElem

HiNum

HiText

ListBoxRefresh

MakeListBox

NoLines

NoTitles

SetListBoxHiElem

SetListBoxTopNum

TopNum

1.139 SetListBoxTabStopsArray function

Function prototype:

```
BOOL SetListBoxTabStopsArray(ListBox *nlb, BOOL refresh, short num, int *tabs);
```

Description:

Set the tab stops in a list box so that data can be displayed in columns. If you want a tabbed list box, you should always set your tab stops before adding any titles or items to the list. Calling SetListBoxTabStopsArray after items have been added to the list won't affect items that were added prior to the SetListBoxTabStopsArray call, only those that are added subsequently.

Parameters:

nlb: A pointer to a FoxGUI list box.

refresh: If TRUE, refresh the list box after setting the tab stops. Since SetListBoxTabStopsArray doesn't currently afffect items and titles that have already been added, there's currently no real reason to set this to TRUE.

num: The number of tab stops to set. Note that no tab stop is necessary for the first column which will always be at the left border of the list box. In other words, if you want to have text in 3 columns, you only need to set 2 tab stops - for the second and third columns.

tabs: An array of tab stops. Each should be the offset for that tab stop in pixels from the left border.

For example, if you have a list box lb in which you want to show three columns of text entitled "Stock no.", "Description" and "In stock", you might set your tab stops and titles as follows:

```
/* Our list box uses an 8 point fixed width font so to allow
   room for a 9 digit stock no. preeceded by a two character
   width gap between columns, we'll put the first tab stop at
   11*8 and then to allow room for a 17 character description
   with the same gap between columns we'll put the second tab
   stop at 30*8. */
int tabs[2];
tabs[0] = 11*8; tabs[1] = 30*8;
SetListBoxTabStopsArray(lb, FALSE, 2, tabs);
AddListBoxTitle(lb, "Stock no.\tDescription\tIn stock");
```

Returns:

TRUE for success, FALSE for failure.

Known bugs:

None.

See also:

AddListBoxItem

AddListBoxTitle

ClearListBoxItems

ClearListBoxTabStops

ClearListBoxTitles

ListBoxRefresh

MakeListBox

1.140 SetListBoxTopNum function

Function prototype:

```
void SetListBoxTopNum(ListBox *lb, int num, BOOL refresh);
```

Description:

Sets the top item number shown in a list box. The user of your application can do this themselves using the scroll bar or buttons on the right hand edge of the list box but this function allows you to set it from within your application code if you require. If a user scrolls the list using the scroll bar or buttons, the Gui will always attempt to keep the list box full e.g. if the list box has 20 lines and 100 elements (the list box can show 20 elements at a time) the user won't be allowed to scroll down below the point where the top element shown is item number 81 (81 to 100 inclusive is 20 items). This function provides no such checking for you and would quite happily set the top item number to 101! However, it is advised that you try to keep to the look and feel of the Gui by not doing so and you can use the function

NoLines

to

help you work out the highest number that you should specify for the top number in the list box. You will have to keep a count of the number of items in the list box yourself.

Parameters:

lb: A pointer to a FoxGUI list box.
num: The item number to set as the top item in the list box.
refresh: If TRUE, refresh the list box to show the list starting with the new top item. Set this to FALSE if you have many other changes to make to the list box - that way, instead of

```
            refreshing after each change you can refresh just once at the
            end.

Known bugs:

   None.

See also:

                ListBoxRefresh

                MakeListBox

                NoLines

                NoTitles

                SetListBoxHiNum

                TopNum
```

1.141 SortListBox function

```
               Function prototype:

   void SortListBox(ListBox *p, int flags, int startnum, BOOL refresh);

Description:

   Sorts the items in a list box (numerically or alphabetically) into
   ascending or descending order.

Parameters:

          p: A pointer to a list box to sort.
      flags: This should be one of ASCENDING or DESCENDING (to sort
             alphabetically), NUM_ASCENDING or NUM_DESCENDING to sort
             numerically.  If none of these are specified then DESCENDING
             is assumed.  When sorting numerically, the function attempts
             to turn the text for each item into a number and sorts using
             those numbers - any items which can't be turned into numbers
             are treated as if they contained the number zero.
   startnum: The item number to start sorting at.  To sort the whole list,
             set this to 1.  Setting this to any number (n) greater than 1
             will cause the first n-1 items to remain exactly where they
             are and the items from n onwards to be sorted.
    refresh: If TRUE, refresh the list to show the items in their new
             order.  Set this to FALSE if you want to make other changes
             before refreshing the list (it's faster to do all of your
             changes and then refresh once than to refresh after each
             change).

Known bugs:
```

```
   None.

See also:

                AddListBoxItem

                ClearListBoxItems

                ListBoxRefresh

                MakeListBox
```

1.142 TopNum function

```
               Function prototype:

   int TopNum(ListBox *lb);

Description:

   Returns the item number of the first item currently displayed in the
   visible portion of a list box.

Parameters:

   lb: A pointer to a FoxGUI list box.

Returns:

   The number of the item cureently at the top of the visible portion of
   the list box.

Known bugs:

   None.

See also:

                HiElem

                HiNum

                HiText

                MakeListBox

                NoLines

                NoTitles
```

1.143 FoxGUI Drop-Down Listbox functions

Drop-down list boxes are likely to be less familier to Amiga users than
many of the other controls in FoxGUI. Most Amiga applications don't use
them but if you have used MUI applications or Windows applications on a PC
then you will recognise them instantly. Drop-down list boxes are like a
cross between edit boxes and list boxes. When you see a drop-down list box
in an application window, what you will see is an edit box with a button
attached to it's right hand end. You cannot edit the contents of the edit
box directly but if you click on the button, a list of options (which looks
exactly like a list box) appears below the edit box. The list can be
scrolled (if it contains more items than can be shown at once) exactly as a
list box can but clicking on an item causes that item to be copied into the
edit box and the list to disappear. In this way a user can choose from a
set of options decided by the programmer. On Amigas running OS 2.0 or
greater, the user can tab (and shift tab) to drop-down list boxes exactly
as with edit boxes. When the user has selected the drop-down list box
(either by tabbing or by clicking in the edit box part with the mouse) the
user can select items in the list by using short-cut keys. For example, if
the list box is selected and the user presses the "t" key, the first item
in the list beginning with the letter t is selected. If no item in the
list begins with the letter t then nothing is selected. If more than one
item begins with a t then pressing t repeatedly will cycle between them.
As with tabbing, short-cut keys are only supported on OS version 2.0 and
above.

The following drop-down listbox functions are currently available :-

AddToDDListBox

AssociateDDListBox

ClearDDListBox

GetDDListBoxID

GetDDListBoxText

MakeDDListBox

MakeSubDDListBox

RemoveFromDDListBox
RestoreDDListBoxStatus

SetDDListBoxPopup

SetDDListBoxText

SortDDListBox
StoreDDListBoxStatus

See also:

Destroy

DisableControl

EnableControl

1.144 AddToDDListBox function

Function prototype:

```
BOOL AddToDDListBox(DDListBox *list, char *str);
```

Description:

Adds an item (a line of text) to a drop-down list box. The items in a drop-down list box don't become visible until the user clicks on the button at the right hand end of the drop-down list box (on OS V37 or above this can also be achieved by clicking in or tabbing to the text part of the drop-down list box and then pressing the space bar).

Parameters:

list: A pointer to a FoxGUI drop-down list box as returned by the

MakeDDListBox
function.

str: A pointer to a NULL terminated text string to add as the next item in the list. The items in a drop-down list box can be sorted into order using the function

SortDDListBox
.

Returns:

TRUE if successful, FALSE otherwise.

Known bugs:

None.

See also:

AssociateDDListBox

ClearDDListBox

MakeDDListBox

MakeSubDDListBox

RemoveFromDDListBox

SortDDListBox

1.145 AssociateDDListBox function

Function prototype:

```
BOOL AssociateDDListBox(DDListBox *l, DDListBox *m);
```

Description:

Describing what this function does would be very tricky without an example so here comes an example:

Let's assume that you have twenty items that you want a user to be able to choose from but you want the user to be able to choose up to five of those items simultaneously. A list box or a drop-down list box only allow you to choose one item at a time so you need another solution. One solution (probably not the most elegant but the one that illustrates the use of this function) is to create five drop-down list boxes, each populated with the same list of items so that the user can choose an item in each. It would be very wasteful to have to populate each drop-down list box with the same set of items which is where this function comes in. AssociateDDListBox causes two drop-down list boxes to become associated with each other. What this means is that they share a list of items. Adding an item to one will then also cause it to be added to the other. Removing an item from one will also cause it to be removed from the other. This is not achieved by automating the process of adding to the other list box - they physically share the same list data which means that you save the extra memory that would otherwise be needed. You can associate as many list boxes together as you like - there is no limit. However, please read the known bugs section at the end of this function definition.

Parameters:

l: A pointer to the destination drop-down list box. This list box must be empty (i.e. have no items in it) for this function to succeed. If the list box already has items in it, you can clear them using the

ClearDDListBox
function.

m: A pointer to the source drop-down list box. It is recommended that you fully populate this list box before making the association but it is only necessary to make sure that it contains one item before the association is made.

Returns:

TRUE for success, FALSE for failure.

Known bugs:

If your source drop-down list box is empty then all sorts of problems could occur. AssociateDDListBox will return TRUE but the list boxes

won't quite behave as though they are associated. At some point I will fix the code to stop this from happening but for the time being it should be fairly simple just to ensure that you add at least one item to your source drop-down list before making the association. Clearing a list box which is associated with others is also a problem - not for the source box itself but for any drop-down list boxes associated with it. Please believe that if the solution to this problem were trivial I would have done it a long time ago but it would be a lot of hard work for what's probably going to be a rarely used function so my priorities lie elsewhere. If you really need this sorting out then hassle me.

See also:

AddToDDListBox

ClearDDListBox

MakeDDListBox

MakeSubDDListBox

RemoveFromDDListBox

1.146 ClearDDListBox function

Function prototype:

```
void ClearDDListBox(DDListBox *l);
```

Description:

Removes all of the items from the specified drop-down list box.

Parameters:

l: A pointer to a FoxGUI drop-down list box.

Known bugs:

This function can cause problems with associated drop-down list boxes. See the AssociateDDListBox function for more details.

See also:

AddToDDListBox

AssociateDDListBox

MakeDDListBox

```
MakeSubDDListBox

RemoveFromDDListBox
```

1.147 MakeDDListBox function

```
Function prototype:

   DDListBox *MakeDDListBox(void *Parent, int x, int y, int len, int buflen,
      int MaxHeight, int id, BOOL (*callfn) (DDListBox*),
      long flags, void *extension);

Description:

   Make a new FoxGUI drop-down list box.

Parameters:

      Parent: A pointer to an open FoxGUI window or frame in which to put
              the new drop-down list box.
           x: The x coordinate in pixels of the left edge of the drop-down
              list box relative to the left edge of the window/frame.
           y: The y coordinate in pixels of the top edge of the drop-down
              list box relative to the top edge of the window/frame.
         len: The length in pixels of the text portion of the list box (A
              drop-down list box looks like an edit box with a button on
              the right hand end which is used to drop the list).  The
              button is a fixed width (17 pixels) so the total length of
              the drop-down list box is len+17.
      buflen: The number of characters to reserve for the text buffer -
              this limits the length of the string that can be shown in the
              text portion of the list box and currently has a maximum
              value of 256 which should be more than enough.  You do not
              need to allow for the NULL terminator - this function will
              automatically allocate one more character than you specify to
              allow for this so you could just set buflen to the strlen()
              of the longest item that you're going to put in the drop-down
              list.
   MaxHeight: The maximum number of lines to show at a time when the list
              box is dropped.  For example, if you set this to 5 and the
              list box contains more than 5 items then only five will be
              visible in the list when you drop it but the list will have
              scroll buttons and a drag-bar to allow you to scroll through
              the other items.  When you create a list box, it will check
              whether there is enough space on the screen below the list
              box in which to drop the box with your specified MaxHeight
              and if not then it will check to see whether there's room
              above the drop-down list box to "drop" the list upwards.  If
              there's not enough room to "drop" the list in either
              direction with your specified MaxHeight then MakeDDListBox
              will fail.  If you are going to make this a "popup" list box
              with the function
                 SetDDListBoxPopup
                 then you can safely set
```

MaxHeight to 0.

id: This parameter doesn't affect the way the drop-down list box looks or works in any way but gets stored as part of the drop-down list box structure and can be found at any time using the
GetDDListBoxID
function. This is used in an entirely analogous way to the id parameter for edit boxes (see the
MakeEditBox
function for details).

callfn: A pointer to a function to call when the user selects an item in a drop-down list box. There are a number of ways a user can do this and hence a number of ways to trigger this function but whichever method the user uses to pick an item, the result is always the same and it isn't important for the function to know how it was triggered. The prototype for the function should be as follows:-

```
short CALLBACK MyDDListBoxFn(DDListBox *lb)
```

The function will be sent a pointer to the drop-down list box but no indication of which item was picked. However, you can find out which item was picked from within the function by calling the
GetDDListBoxText
function. See the
MakeEditBox
function for an example of how to use the id parameter for dealing with arrays of drop-down list boxes from within the call-back function. The return value of this call-back function isn't currently used by FoxGUI but for future compatibility you should return GUI_CONTINUE.

flags: Only three flags are currently available for use with drop-down list boxes:
S_AUTO_SIZE
, THREED and DD_CLEAR.
If the THREED flag is specified then the border around the list box will be drawn (using the Gui pen colours) in such a way as to make it look as though it's pressed into the screen. If the THREED flag is not specified then the border will be a simple box around the edge in the current default border colour. DD_CLEAR specifies that the drop-down list box will be clear (i.e. see-through). In other words, the background colour of the drop-down list box will be the colour of the window or frame in which it was created.

extension: This is reserved for future expansion and should be set to NULL.

Returns:

If successful, a pointer to a new FoxGUI drop-down list box. NULL otherwise.

Known bugs:

None.

See also:

AddToDDListBox

AssociateDDListBox

ClearDDListBox

Destroy

DisableControl

EnableControl

MakeSubDDListBox

RemoveFromDDListBox

SetDDListBoxPopup

SortDDListBox

GetDDListBoxID

GetDDListBoxText

SetDDListBoxText

MakeEditBox

1.148 MakeSubDDListBox function

Function prototype:

```
DDListBox *MakeSubDDListBox(DDListBox *lb, char *string, int left, int top, int ↩
    width,
   int height, int id, BOOL (*callfn)(DDListBox*), void *extension);
```

Description:

Rather like menus which can have sub-menus, so drop-down list boxes can have sub-list boxes. Creating a sub-list box adds an item to a drop-down list box which, when selected, (rather than populating the text part of the drop-down list box) pops up another list of options from which the user must then select an item which will be used to populate the text part of the drop-down list box. Unlike menus, there is no limit to the number of levels a drop-down list box can have i.e. a sub-list box may in turn have items which, when selected open further sub-list boxes.

Most functions which can be applied to drop-down list boxes can also be

applied to sub-list boxes. For example, you add items to a sub-list box using the

AddToDDListBox

function exactly as you do for drop-down list boxes. Sub-list boxes can also be associated using the

AssociateDDListBox

function. By their very nature, sub-list boxes are pop-up so there is no need to use the

SetDDListBoxPopup

function unless you want to later change where the list will pop-up.

Parameters:

lb: A pointer to an existing FoxGUI drop-down list box to which to add the sub-list box.

string: An item to add to the drop-down list box specified in lb which, when selected will trigger the sub-list box. It is a good idea to indicate somehow in the text you supply that this item triggers a sub-list box. I usually use the "»" character (created by pressing Alt-0) at the end of the text to indicate this.

left: Sub-list boxes behave like pop-up list boxes (see

SetDDListBoxPopup

) so they need to be told where to appear on the screen. Like pop-up list boxes, they aren't constrained by the dimensions of the window they are created in but they are constrained by the dimensions of the screen that window appears on. This parameter specifies the left edge of the list when it pops up and is offset from the left edge of the screen (not the window).

top: The top edge of the list when it pops up, offset from the top edge of the screen.

width: The width in pixels of the list when it pops up. If you specify -1 for the width, the Gui will calculate the width necessary for all items in the list to be shown without being truncated and will use that width for the list box when it pops up. The calculation actually takes place when the list box pops up (not when you call this function) so that items added to or removed from the list after calling this function are also taken into account. This does make dropping the list marginally slower especially if there is a large number of items in the list but it means that in an application where the user can choose the font, you don't have to worry about calculating how wide you want the box to be - FoxGUI will work it out for you. Note that if the width is set to -1, the left parameter is ignored because the Gui will have to calculate where to place the left edge of the box so that it will fit on the screen.

height: Rather like the MaxHeight parameter to

MakeDDListBox

, this is the height of the list when it pops up but is specified in lines of text - not pixels.

If the left, top, width and height parameters are such that the

```
           area required will go beyond the boundaries of the screen, the
           function will fail.

       id: See
                 MakeDDListBox
                 or
                 MakeEditBox
                 for details.
   callfn: A pointer to a function to call when an item is selected from
           the new sub-list box.  See the callfn parameter to the function

                 MakeDDListBox
                 for details.
extension: This is reserved for future expansion and should be set to
           NULL.

Returns:

   If successful, a pointer to the new sub-list box is returned.  If not,
   NULL is returned.

Known bugs:

   None.

See also:


                 AddToDDListBox

                 AssociateDDListBox

                 ClearDDListBox

                 Destroy

                 MakeDDListBox

                 RemoveFromDDListBox

                 SortDDListBox

                 GetDDListBoxID
```

1.149 RemoveFromDDListBox function

```
               Function prototype:

   BOOL RemoveFromDDListBox(DDListBox *list, char *str);

Description:

   Remove the specified item from the specified drop-down list box.
```

```
Parameters:

   list: The drop-down list box from which you want to remove an item.
    str: The text of the item to remove.  This must exactly match the text
         given when the item was added to the drop-down list box but
         needn't be the same string.  In other words, the pointers do not
         need to match but the text must.

Returns:

   TRUE for success, FALSE for failure.

Known bugs:

   None.

See also:

                AddToDDListBox

                ClearDDListBox

                MakeDDListBox

                MakeSubDDListBox
```

1.150 SetDDListBoxPopup function

```
               Function prototype:

   BOOL SetDDListBoxPopup(DDListBox *l, int x, int y, int width, int height);

Description:

   Usually when you click on the button on the right hand end of the text
   portion of a drop-down list, the list will drop immediately below the
   button (or immediately above if the drop-down list box is close to the
   bottom of the screen).  The SetDDListBoxPopup allows you to specify
   where the list box will pop up when the button is pressed.

Parameters:

        l: A pointer to the drop-down list box which is to pop up rather
           than drop down.
        x: The distance in pixels between the left edge of the screen and
           the left edge of the area where you want the list to pop up.
        y: The distance in pixels between the top edge of the screen and
           the top edge of the area where you want the list to pop up.

           Note that the x and y parameters are both relative to the screen
           not the window.

    width: The width in pixels of the list when it pops up.  If you specify
```

-1 for the width, the Gui will calculate the width necessary for all items in the list to be shown without being truncated and will use that width for the list box when it pops up. The calculation actually takes place when the list box pops up (not when you call this function) so that items added to or removed from the list after calling this function are also taken into account. This does make dropping the list marginally slower especially if there is a large number of items in the list but it means that in an application where the user can choose the font, you don't have to worry about calculating how wide you want the box to be - FoxGUI will work it out for you. Note that if the width is set to -1, the x parameter is ignored because the Gui will have to calculate where to place the left edge of the box so that it will fit on the screen.

height: The height (in lines of text) of the list when it pops up. This is rather like the MaxHeight parameter to the function

MakeDDListBox

.

Returns:

TRUE for success, FALSE for failure. Note that this function will fail if the area described in the x, y, width and height parameters extends beyond the boundary of the screen.

Known bugs:

None.

See also:

MakeDDListBox

MakeSubDDListBox

1.151 SortDDListBox function

Function prototype:

```
void SortDDListBox(DDListBox *p, int flags);
```

Description:

Sort the items in the specified drop-down list box into the order specified by the flags supplied. This will have no immediately visible effect on the drop-down list box but will affect the order in which the items are shown next time the user drops the list.

Parameters:

p: A pointer to a FoxGUI drop-down list box whose items are to be sorted.

```
   flags: The available flags are NUM_ASCENDING, NUM_DESCENDING,
          ASCENDING, DESCENDING and IGNORE_CASE.  ASCENDING and DESCENDING
          specify that sorting should be alphabetic and should be ascending
          or descending respectively.  NUM_ASCENDING and NUM_DESCENDING
          specify that the sorting should be numeric - i.e. treat the text
          as numbers.  The difference is that alphabetically, the string
          "02" would come before the string "1" whereas if the strings were
          converted to numbers, the correct numerical order would be
          obtained.  The IGNORE_CASE flag, when combined with either of the
          alphabetic flags ASCENDING or DESCENDING causes the case of the
          characters to be ignored when sorting (without this, the entire
          alphabet of upper case characters is considered to come before
          any of the lower case characters).  If no flags are specified,
          the DESCENDING flag is assumed.

Known bugs:

   None.

See also:

                AddToDDListBox

                MakeDDListBox

                MakeSubDDListBox
```

1.152 GetDDListBoxID function

```
               Function prototype:

   int GetDDListBoxID(DDListBox *l)

Description:

   Returns the id of the specified drop-down list box, as supplied to

                MakeDDListBox
                or
                MakeSubDDListBox
                when the drop-down list box was
   created.  See
                MakeDDListBox
                or
                MakeEditBox
                for details.

Parameters:

   l: A pointer to the FoxGUI drop-down list box whose id you want to know.

Returns:
```

The id of the specified drop-down list box.

See also:

GetEditBoxID

MakeDDListBox

MakeEditBox

MakeSubDDListBox

1.153 GetDDListBoxText function

Function prototype:

```
char *GetDDListBoxText(DDListBox *l)
```

Description:

Returns a pointer to a NULL terminated string containing the text currently displayed in the text portion of a drop-down list box. Passing a sub-list box as a parameter to this function would be meaningless.

Parameters:

l: A pointer to the drop-down list box whose current text you want to know.

Returns:

A pointer to a NULL terminated text string containing the text currently shown in the text portion of the drop-down list box. The pointer returned is a pointer to the actual buffer used by the drop-down list box so you should never directly modify this string in any way. If you keep a copy of the pointer you should also remember that it will become invalid when the drop-down list box is destroyed. The safest thing to do is make your own copy of the string using a function such as strcpy. If this function fails, NULL will be returned. If there is no text in the box then an empty string will be returned. Remember that the text may not necessarily match an item in the drop-down list box itself. This will happen if no item has been selected, the item has been selected from a sub-list box, the item selected has since been removed from the list or the text has been set to a non-matching value by calling the function
SetDDListBoxText
.

See also:

GetEditBoxText

MakeDDListBox

MakeSubDDListBox

RemoveFromDDListBox

SetDDListBoxText

1.154 SetDDListBoxText function

Function prototype:

```
BOOL SetDDListBoxText(DDListBox *l, char *c)
```

Description:

Set the text shown in the text portion of a drop-down list box. Note that the text need not match any of the items available in the list for selection by the user although I can't really see why you would want to do that.

Parameters:

l: A pointer to the drop-down list box whose text you want to set.
c: A pointer to a NULL terminated text string containing the text you wish to put in the text portion of the drop-down list box. A copy will be made of the string supplied so there is no need to preserve the string passed after calling the function.

Returns:

TRUE for success, FALSE for failure.

See also:

SetEditBoxText

MakeDDListBox

MakeSubDDListBox

GetDDListBoxText

1.155 FoxGUI Outputbox functions

Output boxes provide a convenient way of placing text or data onto a FoxGUI window. If you want a piece of static text placed in a window (for example

a title or label of some description) you can make an output box at that position and then set the output box text using one of the functions below. Alternatively, if you want to display a piece of data that may change during the time the program is running, an output box is perfect for that too. Just make the output box and then use one of the functions below to set the output box text each time it needs to change. As with edit boxes, TEXT, INT and FLOAT types are supported and the number of decimal places for FLOAT types can be specified.

The following outputbox functions are currently available :-

MakeOutputBox

SetOutputBoxCols

SetOutputBoxDP

SetOutputBoxDouble

SetOutputBoxInt

SetOutputBoxText

GetOutputBoxID

See also:

Destroy

Hide

Show

1.156 MakeOutputBox function

Function prototype:

```
OutputBox *MakeOutputBox(void *Parent, int x, int y, int width, int len,
   int id, char *InitialValue, long flags, void *extension);
```

Description:

Create a new output box in the specified window or frame.

Parameters:

Parent: A pointer to the FoxGUI window or frame in which to create the output box.

x: The coordinate of the left edge of the new output box relative to the left hand edge of the specified window/frame.

y: The coordinate of the top edge of the new output box relative to the top edge of the specified window/frame.

Note that this is from the very top of a window so a y coordinate of 0 will cause the output box to be at least partly obscured by the window's title bar if it has one.

width: The width of the output box in pixels. This width includes the border if one is specified (see flags below).

len: The maximum length (in characters) of text that can be shown in the output box. This will be used to allocate memory for the text buffer so you should try not to set this higher than you need it. This number cannot be more than 256.

id: Sets the id of the output box which can be retrieved at any time with the GetOutputBoxID function. The id for an output box is used in exactly the same way as the id of an edit box. See MakeEditBox for a full description.

InitialValue: A string containing text to show in the outputbox. Specifying a non-NULL value here is equiuvalent to setting this to NULL and then immediately calling SetOutputBoxText

flags: Currently seven flags are available for use with output boxes: THREED, NO_BORDER, JUSTIFY_LEFT, JUSTIFY_CENTRE, JUSTIFY_RIGHT, S_AUTO_SIZE and S_FONT_SEMNSITIVE. If the NO_BORDER flag is specified then the output box will not have a border. If the THREED flag is specified then the output box will have a three dimensional border drawn in the current Gui pens (these can be modified by calling

SetGuiPens

). If neither flag is specified then the output box will have a two dimensional border in the current default border colour. The three JUSTIFY_ flags are mutually exclusive. At most one of them should be specified. If the JUSTIFY_LEFT flag is set then any text placed in the output box with the functions

SetOutputBoxText

,

SetOutputBoxDouble

or

SetOutputBoxInt

will begin at the left edge of the output box. If JUSTIFY_CENTRE is specified then text will appear centred within the bounds of the output box and if JUSTIFY_RIGHT is selected then the right edge of the text will be aligned with the right edge of the output box. If none of the JUSTIFY_ flags is selected then JUSTIFY_LEFT is used by default. Starting with release 2.0, whichever justification method is selected, the text will be clipped appropriately to ensure that it never extends beyond the border of the output box. If the output box is resized (due to a window resizing) then some of the

```
                clipped text may become visible or more may be clipped.
                If the S_FONT_SENSITIVE flag is specified then the output
                box width and height are set according to font size and
                caption.  Output boxes with this flag set will still resize
                when in a resizable window because their contents may
                change and so resizing may still be important.
      extension: This is reserved for future expansion and should be set to
                NULL.

Returns:

   If successful, a pointer to the new output box, otherwise NULL.

Known bugs:

   None.

See also:

                 Destroy

                 DisableControl

                 EnableControl

                 Hide

                 Show

                 SetOutputBoxCols

                 SetOutputBoxDP

                 SetOutputBoxDouble

                 SetOutputBoxInt

                 SetOutputBoxText

                 GetOutputBoxID

                 WriteText
```

1.157 SetOutputBoxCols function

```
                Function prototype:

   void SetOutputBoxCols(OutputBox *ob, int Bcol, int Tcol, BOOL refresh);

Description:

   Modify the colours of an existing FoxGUI output box.
```

```
Parameters:

        ob: A pointer to the output box whose colours are to be modified.
   refresh: If TRUE, redraw the output box in its new colours.

   The Bcol and Tcol parameters are identical to the Bcol and Tcol
   parameters passed to
                MakeOutputBox
               .

Known bugs:

   None.

See also:


                MakeOutputBox

                SetOutputBoxDP

                SetOutputBoxDouble

                SetOutputBoxInt

                SetOutputBoxText
```

1.158 SetOutputBoxDP function

```
               Function prototype:

   void SetOutputBoxDP(OutputBox *p, int dp);

Description:

   Set the number of decimal places which will be shown when a floating
   point (non integral) number is shown in the output box specified.  This
   function will not affect text placed in the output box using the
   functions
                SetOutputBoxInt
                and
                SetOutputBoxText
               .  Only text set
   using the function  SetOutputBoxFloat  is affected.

Parameters:

    p: A pointer to the output box.
   dp: The number of figures to show after the decimal point when floating
       point numbers are displayed.

Known bugs:

   None.
```

See also:

MakeOutputBox
SetOutputBoxDouble

1.159 SetOutputBoxDouble function

Function prototype:

```
void SetOutputBoxDouble(OutputBox *p, double num);
```

Description:

This function converts the double-precision floating point number passed to it into text and places that text in the output box specified. You might use this to display the result of a calculation in a window. If you want to limit the number of decimal places shown in the output box without having to round the number, you can do so by calling the function
SetOutputBoxDP
. The justification specified in the flags parameter of
MakeOutputBox
is respected by this function.

Parameters:

p: A pointer to the output box whose text is to be set or replaced.
num: The number to place in the output box.

Known bugs:

None.

See also:

MakeOutputBox

SetOutputBoxDP

SetOutputBoxInt

SetOutputBoxText

1.160 SetOutputBoxInt function

```
                Function prototype:

   void SetOutputBoxInt(OutputBox *p, int num);

Description:

   This function converts the integral number passed to it into text and
   places that text in the output box specified.  You might use this to
   display the result of a calculation in a window.  The justification
   specified in the flags parameter of
                MakeOutputBox
                is respected
   by this function.

Parameters:

     p: A pointer to the output box whose text is to be set or replaced.
   num: The number to place in the output box.

Known bugs:

   None.

See also:


                MakeOutputBox

                SetOutputBoxDouble

                SetOutputBoxText
```

1.161 SetOutputBoxText function

```
                Function prototype:

   void SetOutputBoxText(OutputBox *p, char *text);

Description:

   Set or modify the text of the specified output box to the text passed to
   this function.  The justification specified in the flags parameter of

                MakeOutputBox
                is respected by this function.

Parameters:

      p: A pointer to the output box whose text is to be set or replaced.
   text: A pointer to the NULL terminated string to be placed into the
         output box.  After calling the function SetOutputBoxText, it is
         not necessary to maintain the text string passed - the function
         will make it's own copy of the string.
```

Known bugs:

None.

See also:

MakeOutputBox

SetOutputBoxDouble

SetOutputBoxInt

1.162 GetOutputBoxID function

Function prototype:

```
int GetOutputBoxID(OutputBox *o)
```

Description:

Returns the id of the specified output box. Output box ids are entirely analogous to edit box ids, a full description of which can be found in the function
MakeEditBox
.

Parameters:

p: A pointer to the output box whose id you want to know.

Returns:

The id of the output box specified.

See also:

MakeOutputBox

1.163 FoxGUI Tab Controls

Tab controls, like frames can be holders of other controls. It's difficult to explain to an Amiga user what a tab control is like because I've never seen them used in Amiga programs before. PC users may have seen them used in the options dialogue in Microsoft Word for Windows 6 or 7. Basically a tab control is a frame with a row of buttons along the top edge. Clicking on any of the buttons will change the contents of the frame. You may, for

example, have a program that is highly configurable and hence has a lot of options the user can choose from. Let's say, for example that the user can change the screen mode that the program runs in, the pallette used for the screen and several other program specific options. The number of controls requiured to display all of these options may well be more than you can easily (or neatly) fit into a window (especially if the program is allowed to run in the basic Amiga Pal modes) so the answer is a tab control. You create a tab control with three frames (and hence three buttons). The frames in a tab control are always the same size as each other and directly on top of each other so that you only see one frame at a time. The first frame could contain the screen mode preferences, the second the pallette preferences and the third all of the other options. The three buttons could then be labelled "Screen", "Pallette" and "Other". When you create controls in a tab control frame you use the normal "Make" functions (MakeButton, MakeEditBox etc) but rather than passing a pointer to a GuiWindow as the first parameter, you pass a pointer to the frame in the tab control (see the
TabControlFrame
function). You do not need to write
any code to handle what happens when the user clicks the buttons along the top edge of the tab control - this is automatically handled for you by FoxGUI. When you click the first button, the contents of the first frame becomes visible. When you click the second button, the contents of the second frame becomes visible and the contents of the frame shown previously becomes invisible.

Note that any type of control can be created in the frame of a tab control (including frames and further tab controls!)

Rather than try in vain to describe this further, I'll leave it to you to look at the "Characters" program available for download from my web page (see
Suggestions
section for the address) which makes use of tab
controls.

The following tab control functions are currently available :-

MakeTabControlArray

TabControlFrame
See also:

Destroy

DisableControl

EnableControl

1.164 MakeTabControlArray function

Function prototype:

```
TabControl *MakeTabControlArray(void *Parent, int left, int top, int width, int ↩
    height,
   int tabheight, short flags, int *tabwidth, char **caption,  ↩
      TabControlExtension *ext);
```

Description:

Create a new tab control.

Parameters:

Parent:	A pointer to the window or frame in which to create the new tab control.
left:	The left edge (in pixels) of the tab control relative to the left edge of the parent window/frame.
top:	The top edge (in pixels) of the row of buttons (tabs) which runs along the top edge of the tab control, relative to the top of the parent window/frame.
width:	The width (in pixels) of the tab control.
height:	The height (in pixels) of the framed part of the tab control. The overall height of the tab control will be height+tabheight.
tabheight:	The height (in pixels) of the tabs along the top edge of the control.
flags:	The following flags are currently available for tab controls :- TC_CLEAR, S_AUTO_SIZE. The TC_CLEAR flag causes the background colour of the tab control to be the colour of the window or frame in which it was created. The S_AUTO_SIZE flag causes the tab control to get resized if the parent (window or frame) is resized.
tabwidth:	An array of numbers specifying the width of each of the tabs (the buttons along the top edge of the tab control). The final entry in the array should be 0 to indicate that the end has been reached. E.g. if you wanted three tabs with widths of 20, 30 and 40 pixels respectively then you would pass an array of integers containing 20, 30, 40 and 0.
caption:	A pointer to an array of text strings containing the text to appear in the tabs (the buttons along the top edge of the tab control). This array should contain one fewer entries than the tabwidth array because there is no need for an extra entry to indicate the end of the array. In other words, if you want three tabs then you pass an array of three strings. For example, to create three tabs each of width 70 pixels with the captions "Screen", "Pallette" and "Other", the tabwidth and caption arrays would be defined as follows: int tabwidth[4] = { 70, 70, 70, 0 }; char *caption[3] = {"Screen", "Pallette", "Other" };
ext:	This is reserved for future expansion and should be NULL.

Returns:

If successful, a pointer to the new tab control. Otherwise NULL.

Known bugs:

```
   None.

See also:

                Destroy

                TabControlFrame
```

1.165 TabControlFrame function

```
               Function prototype:

   Frame *TabControlFrame(TabControl *tc, int frameno);

Description:

   Controls can be created in windows or in frames.  A tab control is
   really just a series of frames so to create a control in a tab control,
   you first need to get a pointer to the frame.  This function returns a
   pointer to a frame in a tab control.

Parameters:

        tc: A pointer to a tab control.
   frameno: The frame number of the frame you want a pointer to.  Note that
            the frame numbers start at zero so if you have a tab control
            with seven tabs, the frames will be numbered 0 - 6.

Returns:

   If successful, a pointer to a frame within a tab control.  Otherwise
   NULL.  Note that if frameno is more than the number of frames in the tab
   control minus one (frames are numbered starting from zero) then this
   function will return NULL.

Known bugs:

   None.

See also:

                MakeTabControlArray
```

1.166 FoxGUI Miscelaneous functions

```
               The following miscelaneous functions aren't directly related to ↩
                  any type of
FoxGUI control but are still very important.  It is worth browsing the
```

other functions below because they could save you a lot of time and effort
in developing equivalents yourself.

The following miscelaneous functions are currently available :-

CheckMessages

Destroy

DestroyM

DisableControl

DisableM

DrawLines

EnableControl

EnableM

GetDefaultFontCopy

GetWindow

GuiGetLastErr

GuiLoop

GuiMalloc

GuiMessage

GuiTextLength

Hide

IntuiWindow

LibVersion

RegisterGadget

SetDefaultCols

SetDefaultFont

SetDelay

SetGuiPens

SetGuiPensFromPubScreen

SetPeriod

SetPostText

```
                SetPreText

                Show

                UnRegisterGadget

                WriteText
               The following miscelaneous operations are currently defined as  ↩
                  macros :-


                GuiFree
```

1.167 CheckMessages function

```
               Function prototype:

   void CheckMessages(void);

Description:

   Tells FoxGUI to check for any pending messages and respond to them
   immediately.  See the section on
                Multi-threading
                for more information.

Known bugs:

   None.

See also:


                Multi-threading
```

1.168 Destroy function

```
               Function prototype:

   void Destroy(void *Control, BOOL refresh);

Description:

   Destroys the specified control.  This function destroys any control
   including windows and screens.  The only exceptions are menus, menu
   items and sub-menu items.

Parameters:
```

```
   Control: A pointer to the control to destroy.
   refresh: If TRUE, refresh the parent window so that the control is
            visibly removed from the window.  If you are in the process of
            destroying all of the controls in a window in order to close
            the window then you might want to set this to FALSE which will
            make the process faster.  For certain controls (such as windows
            and screens) this parameter is ignored and a refresh occurs.

Known bugs:

   Can't be used to destroy menus, menu items or sub-menu items.

See also:

                DisableControl

                EnableControl

                GetWindow

                Hide

                Show
```

1.169 DisableControl function

```
               Function prototype:

   void DisableControl(void *Control, BOOL refresh);

Description:

   Disables the specified control.  This function works for any control
   except menus, menu items, sub-menu items, output boxes, screens and
   progress bars (disabling output boxes and progress bars would be pretty
   meaningless since the user can't interact with them anyway).  Passing a
   pointer to a GuiWindow is equivalent to calling
                SleepPointer
                  and passing a pointer to a timer is equivalent to calling
                PauseTimer
               .

Parameters:

   Control: A pointer to the control to disable.
   refresh: If TRUE, the control will be redrawn to appear disabled. For
            controls which have no imagery (such as timers) and for certain
            other controls (such as windows) this parameter is ignored.

Known bugs:

   None.
```

```
See also:

                Destroy

                EnableControl

                GetWindow

                Hide

                Show
```

1.170 DrawLines function

```
Function prototype:

   void DrawLines(GuiWindow *win, short *points, int count, int col);

Description:

   Draw straight line(s) in the specified FoxGUI window.

Parameters:

      win: A pointer to a FoxGUI window in which to draw the line(s).
   points: A pointer to an array of points describing the lines to be
           drawn.  Each pair of numbers is treated as the x and y
           coordinates of a point relative to the top left of the specified
           window.  Lines are drawn from the first point given to the
           second, from the second point to the third etc until lines have
           been drawn between count sets of coordinates.
    count: The number of points between which to draw lines.  The points
           array should contain count sets of coordinates (i.e. count * 2
           numbers).
      col: The colour in which to draw the lines.

Known bugs:

   None.
```

1.171 EnableControl function

```
               Function prototype:

   void EnableControl(void *Control, BOOL refresh);

Description:

   Enables the specified control.  This function works for any control
   except menus, menu items, sub-menu items, output boxes, screens and
```

```
   progress bars (enabling output boxes and progress bars would be pretty
   meaningless since the user can't interact with them anyway).
   Passing a pointer to a GuiWindow is equivalent to calling
                WakePointer
                  and passing a pointer to a timer is equivalent to calling
                UnpauseTimer
               .

Parameters:

   Control: A pointer to the control to enable.
   refresh: If TRUE, the control will be redrawn to appear enabled. For
            controls which have no imagery (such as timers) and for certain
            other controls (such as windows) this parameter is ignored.

Known bugs:

   None.

See also:


                Destroy

                DisableControl

                GetWindow

                Hide

                Show
```

1.172 GetWindow function

```
               Function prototype:

   GuiWindow *GetWindow(void *Control);

Description:

   Returns a pointer to the window in which the specified control was
   created.  If the control was created in a frame or a tab control then
   a pointer to the window in which the frame or tab control was created is
   returned.  Similarly if the control is in a frame which itself is in a
   frame which is in a tab control then a pointer to the window in which
   the tab control was created is returned.

Parameters:

   Control: A pointer to the control.  Note that the control may not be a
            menu, menu item, sub-menu item or screen.

Returns:
```

A pointer to the GuiWindow in which the control was created.

Known bugs:

Doesn't work for menus, menu items or sub-menu items.

See also:

Destroy

DisableControl

EnableControl

Hide

Show

1.173 GuiLoop function

Function prototype:

```
void GuiLoop(void);
```

Description:

The GuiLoop function is where all of the Gui events are processed. You call it when you have set up all of the screens/windows/controls that you want the user to have access to when they first run your application and it handles all of the input events for the controls you have set up. When you create a control, you usually supply a pointer to a function as a parameter - for example when you create a button using the MakeButton function, one of the parameters is a pointer to a function to call when the button is clicked. It is the GuiLoop function which ensures that the correct functions are called when these events occur. These call-back functions can themselves create and destroy controls and when the call-back function returns it generally has to return one of two values - one specifying that GuiLoop should continue to process events for your existing windows and controls (and also for any new controls that you created from within the call-back function) and one that informs the GuiLoop function that you're all done and will cause GuiLoop to return.

Known bugs:

None.

See also:

EndGui
InitGui

1.174 GuiMalloc function

```
                Function prototype:

   void *GuiMalloc(unsigned long NoOfBytes, unsigned long flags);

Description:

   This function allocates memory.  It is the function used internally
   within FoxGUI to allocate memory whenever FoxGUI needs it.  I have made
   it available externally because it provides a little bit of control over
   memory allocation which is not available through other means.  Memory
   allocated by the GuiMalloc function should be free'd with the
                GuiFree
                  macro.

   GuiMalloc works in one of two ways depending upon whether the
   FAST_MALLOCS flag was specified in the call to  InitGui .  One provides
   error checking code but is slow by comparison to the other which does
   not.  It is suggested that you use the slower method while developing
   your programs (by passing 0 as the flags parameter to InitGui) and then
   when they are bug free and ready to release, change the call to InitGui
   to pass FAST_MALLOCS as the flags parameter to speed things up by
   removing the error checking code.

   If FAST_MALLOCS has not been specified, GuiMalloc works as follows:
   When you ask GuiMalloc to allocate some memory, it actually allocates
   extra bytes at the start and end of the memory you asked for and puts
   special characters in these bytes along with a record of the number of
   bytes allocated.  Then, when you call GuiFree with a pointer to some
   memory to free it checks for those special characters immediately before
   and after the memory you allocated and returns an error if they are not
   there.  This prevents you from attempting to free memory that you haven't
   allocated and memory that has become corrupted.  If memory has become
   corrupted it's usually due to something like writing beyond the bounds
   of an array.  In general, you wouldn't find out about this until your
   Amiga crashed due to some processes memory having been scribbled on by
   itself or another process but by using GuiMalloc and GuiFree there's a
   chance that you will be allerted to it sooner.  Of course, this sort of
   protection isn't as good as that provided by running Enforcer and
   Mungwall but it doesn't require an MMU (which Enforcer does) and if a
   call to GuiFree fails, the Gui will tell you exactly which call failed
   by telling you the filename and line number of the offending GuiFree
   call.

Parameters:

   NoOfBytes: The number of bytes of memory to allocate.
       flags: At the moment, the only available flag is MEMF_CLEAR which
              will cause all of the allocated memory to be set to zeros if
              specified.

Returns:

   A pointer to the allocated memory or NULL for failure.  It is essential
   that you check the return code of this function before using the memory
```

```
   - writing over NULL memory is very likely to cause a crash!

Known bugs:

   None.

See also:

    InitGui

                GuiFree
```

1.175 GuiMessage function

```
               Function prototype:

   short GuiMessage(void *Scr, char *text, char *title, int detail, int block,
      int flags);

Description:

   Display a message to the user modally and get the user's response.  This
   function opens a window on the specified screen.  The window displays
   the specified message and contains one or more buttons which can be
   specified in the flags parameter.  The return value of the function
   depends upon which button the user presses to respond to the message.
   The message is modal so the user will not be able to activate or
   interact with any FoxGUI controls in any other windows while the message
   is shown.

Parameters:

      Scr: A pointer to a FoxGUI screen (or the name of a public screen)
           on which to display the message.
     text: The text of the message.  The text may contain linebreak
           characters (specified as '\n' in C) to display multi-line
           messages.
    title: A short text string to show in the title bar of the message
           window.
   detail: The pen colour to use for the windows detail pen and for the
           message text.
    block: The pen colour to use for the windows block pen.
           These are equivalent to the Dpen & Bpen parameters passed to the

                 OpenGuiWindow
                 function.
    flags: The flags determine the buttons and images shown in the window.
           A combination of the following flags should be used to determine
           which buttons are shown: GM_OKAY, GM_YES, GM_NO, GM_CANCEL,

           All buttons appear along the bottom edge of the window and are
           spaced such that the window appears balanced.  The exact location
           of each button will depened on how many other buttons are visible.
```

```
            Specifying GM_OKAY on its own will give the message window a
            single button labelled "Okay".  This is useful for messages
            giving the user some information where they merely have to
            respond to proceed.

            Specifying GM_OKAY and GM_CANCEL together will give the window
            two buttons labelled "Okay" and "Cancel".  This might be useful
            when the user has just told your application to erase the entire
            contents of your hard-disk and you want to get confirmation
            before proceeding.  Specifying GM_YES in combination with GM_NO
            gives a similar effect but the buttons are labelled "Yes" and
            "No" instead of "Okay" and "Cancel".

            Specifying GM_YES, GM_NO and GM_CANCEL together will give the
            window three buttons labelled "Yes", "No" and "Cancel".  This
            gives the user alternative methods of proceeding or the option
            not to proceed.

            All of the buttons have hot-keys, the hot-key being the first
            letter of the buttons caption in each case.  Also, the Yes and
            Okay buttons can be activated by the return key and the No and
            Cancel buttons can be activated by the escape key.  If the No
            and Cancel buttons are both present in the window then the
            escape key will activate the Cancel button.

            The flags GM_INFORMATION, GM_EXCLAMATION, GM_QUESTION, GM_X,
            GM_STOP and GM_CROSSBONES cause an image to be displayed at
            the left of the message window.

Returns:

   Returns either GM_OKAY, GM_YES, GM_NO or GM_CANCEL depending on the
   response selected by the user.  If the function fails to open the window
   and display the message, it will return -1 or 0 immediately.

Known bugs:

   None.
```

1.176 GuiTextLength function

```
Function prototype:

   long GuiTextLength(char *text, struct TextAttr *font);

Description:

   Returns the length in pixels of the specified string using the specified
   font.

Parameters:

   text: A text string to find the length of.
   font: A pointer to an intuition font to use when working out the string
```

```
         length.  Note that if font is NULL then FoxGUI will use the
         current default Gui font which may not be the font you wish to use
         when rendering the text.

Returns:

   The length of the string in pixels.

Known bugs:

   None.
```

1.177 Hide function

```
               Function prototype:

   void Hide(void *Control);

Description:

   Hides the specified control, removing it's imagery from the window,
   frame or tab control which contains it.  When a control is hidden it can
   still be updated (for example,
               SetEditBoxText
               still works when an edit
   box is hidden) but changes will not be seen until the control is shown
   again (using the
               Show
               function).  The user of your application will
   not be able to interact with a hidden control in any way.  If a frame or
   tab control is hidden, any controls created within it will also be
   hidden.  These controls will become visible again when the frame/tab
   control is shown unless they have been hidden independantly before or
   since the parent control was hidden.  It is never possible to make a
   control visible if it is in a hidden frame or tab control.

Parameters:

   Control: A pointer to the control to be hidden.

Known bugs:

   Screens, windows, timers, menus, menu items and sub-menu items cannot be
   hidden.  (Timers have no imagery so in effect they are always hidden).

See also:


               Show
```

1.178 IntuiWindow function

```
                Function prototype:

   struct Window *IntuiWindow(GuiWindow *gw);

Description:

   This function returns a pointer to the intuition Window structure used
   by the specified FoxGUI window.  It is mainly of use when adding
   non-FoxGUI gadgets to a FoxGUI window.

Parameters:

   gw: A pointer to the FoxGUI window whose intuition Window you require a
       pointer to.

Returns:

   If gw is a pointer to a FoxGUI window then a pointer to the relevant
   intuition window is returned, otherwise NULL is returned.

Known bugs:

   None.

See also:


                RegisterGadget

                UnRegisterGadget
```

1.179 LibVersion function

```
Function prototype:

   short LibVersion(void);

Description:

   Since calling OpenLibrary for the FoxGUI library opens the intuition
   library for you, this function allows you to find out what version of
   the intuition library was opened.  The return value of this function
   will never be less than 33 since that is the minimum requirement for
   FoxGUI.

Returns:

   The version number of the intuition library opened by InitGui.

Known bugs:

   None.
```

1.180 RegisterGadget function

```
                Function prototype:

   BOOL RegisterGadget(struct Gadget *gad, GuiWindow *gadwin,
      int (*gadfn)(struct gadget*, struct IntuiMessage *));

Description:

   Registers a user-created intuition gadget with FoxGUI so that FoxGUI can
   inform the user (by calling a user-defined function) whenever an
   intuition message is received which relates to that gadget (for example
   when the gadget is clicked on with the mouse).

Parameters:

      gad: A pointer to the intuition gadget to register (note that this
           function can be called before or after calling the intuition
           function AddGadget to add the gadget to the window's gadget
           list).
   gadwin: A pointer to the FoxGUI window which contains (or is about to
           contain) the gadget.  If the gadget is created in a non-FoxGUI
           window then this parameter should be NULL.
    gadfn: A pointer to the function to call whenever the gadget gets an
           intuition message (an IntuiMessage).  The prototype should be as
           follows:

           int CALLBACK MyFunction(struct Gadget *gad, struct
              IntuiMessage *message)

           When the function is called it will be passed a pointer to the
           gadget to which the IntuiMessage refers and a pointer to the
           IntuiMessage itself (note that this is a pointer to the actual
           IntuiMessage that intuition sent to FoxGUI so it is important
           that you don't modify it in any way).  It is your responsibility
           to reply to the message (with the intuition function ReplyMsg)
           when you have finished with it - FoxGUI doesn't do this for you.
           Your function should return either
                 GUI_CONTINUE or GUI_END
                .

Returns:

   TRUE if successful, FALSE otherwise.

Known bugs:

   None.

See also:


                IntuiWindow

                UnRegisterGadget
```

1.181 SetDelay function

```
                Function prototype:

   void SetDelay(int time);

Description:

   When the user clicks on a pushbutton which was created with the BN_AR
   flag (so that the button will be triggered continually while it is held
   down), the button will be triggered immediately and then there will be a
   DELAY before it is next triggered, after which there will be a regular
   PERIOD between subsequent triggers.  This function sets that initial
   delay.

Parameters:

   time: The time (in milliseconds) to delay after the initial triggering
         of auto-repeating pushbuttons before triggering again.

Known bugs:

   None.

See also:


                 SetPeriod
```

1.182 SetGuiPens function

```
Function prototype:

   void SetGuiPens(short hipen, short lopen);

Description:

   Sets the pen colours used by FoxGUI to render 3D borders.  The Gui uses
   two colours - one bright and one dark to make the objects appear as
   though light were being shone across the screen from a light source at
   the top left.  Hence, objects which stand out from the screen will have
   a bright top left hand corner and a dark (shaddowed) bottom right
   corner.  Objects which appear pressed into the screen will have a
   shaddowed top left corner and a bright bottom right corner.

Parameters:

   hipen: A bright colour used to draw well lit edges (1 by default).
   lopen: A dark colour used to draw shaddowed edges (2 by default).

   The default values shown are used if this function is not called by your
   application.
```

Known bugs:

None.

1.183 SetGuiPensFromPubScreen function

Function prototype:

```
BOOL SetGuiPensFromPubScreen(char *pub_screen_name);
```

Description:

Like the SetGuiPens function, this function sets the pen colours used by FoxGUI to render 3D borders. Unlike SetGuiPens where you pass the pen colours to use, this function takes the name of a public screen as a parameter and sets the Gui pens to the same values as used by the public screen named. This function only works on Amigas which support public screens.

This function is particularly useful for picking up the pen colours from the workbench screen (the public name of the workbench screen is "Workbench"). This ensures that whatever the users system colours are, the FoxGUI gadgets will appear in the same colours and hence in 3D.

Parameters:

pub_screen_name: The name of the public screen whose pens are to be copied (public screen names are case sensitive).

Returns:

TRUE if successful, FALSE otherwise.

Known bugs:

None.

1.184 SetPeriod function

Function prototype:

```
void SetPeriod(int time);
```

Description:

When the user clicks on a pushbutton which was created with the BN_AR flag (so that the button will be triggered continually while it is held down), the button will be triggered immediately and then there will be a DELAY before it is next triggered, after which there will be a regular

```
   PERIOD between subsequent triggers.  This function sets that period.

Parameters:

   time: The time (in milliseconds) to delay between each triggering of the
         auto-repeating pushbutton.

Known bugs:

   None.

See also:

                 SetDelay
```

1.185 SetPreText function

```
               Function prototype:

   OutputBox *SetPreText(Widget *Parent, char *text);

Description:

   Creates a pre-text lable for a control (i.e. prints the specified text
   string to the left of the specified control).  The label is stored as
   part of the control and will be automatically hidden if the control is
   hidden or destroyed if the control is destroyed.  It is not necessary,
   therefore to keep the return value of SetPreText.  The label is actually
   created as a borderless OutputBox but calling Destroy() and passing a
   pointer to this OutputBox will not work - the only way to destroy the
   pre-text label is to destroy the control it is attached to.

Parameters:

   Parent: A pointer to the control to which the text will be attached.
           This can be any type of control except a Window, a Screen,
           a Frame or a Menu.
     text: The text to print to the left of the control.

Returns:

   A pointer to an OutputBox.  However, there is little point in keeping
   the return value of this function as discussed above.

See also:

                 SetPostText
```

1.186 SetPostText function

```
                Function prototype:

   OutputBox *SetPostText(Widget *Parent, char *text);

Description:

   Creates a post-text lable for a control (i.e. prints the specified text
   string to the right of the specified control).  The label is stored as
   part of the control and will be automatically hidden if the control is
   hidden or destroyed if the control is destroyed.  It is not necessary,
   therefore to keep the return value of SetPostText.  The label is actually
   created as a borderless OutputBox but calling Destroy() and passing a
   pointer to this OutputBox will not work - the only way to destroy the
   post-text label is to destroy the control it is attached to.

Parameters:

   Parent: A pointer to the control to which the text will be attached.
           This can be any type of control except a Window, a Screen,
           a Frame or a Menu.
     text: The text to print to the right of the control.

Returns:

   A pointer to an OutputBox.  However, there is little point in keeping
   the return value of this function as discussed above.

See also:


                 SetPreText
```

1.187 Show function

```
                Function prototype:

   void Show(void *Control);

Description:

   Shows the specified hidden control.  Controls are automatically shown
   when they are first created so this function is only used to show
   controls which have subsequently been hidden using the
                Hide
                function.
   If a frame which contains other controls is shown then any controls
   contained within the frame will also be shown unless they themselves
   have been hidden either before or after the frame was hidden.  The same
   is true for tab controls.  For example, calling Show for a hidden
   control in a hidden frame won't cause the control to be shown
   immediately because the parent frame is still hidden but if the parent
```

frame is now shown (using the Show function again) then the frame and the control will become visible.

Parameters:

Control: A pointer to the hidden control to show.

Known bugs:

Screens, windows, timers, menus, menu items and sub-menu items cannot be hidden (and hence shown). (Timers have no imagery so in effect they are always hidden).

See also:

Hide

1.188 UnRegisterGadget function

Function prototype:

```
BOOL UnRegisterGadget(struct Gadget *gad);
```

Description:

Removes a user-defined gadget from FoxGUI's registry. This function should only be used with gadgets that have previously been registered using the RegisterGadget function. You should always remove a gadget from FoxGUI's registry when you destroy it.

Parameters:

gad: A pointer to the intuition gadget to remove from the registry.

Returns:

TRUE if successful, FALSE otherwise.

Known bugs:

None.

See also:

RegisterGadget

1.189 WriteText function

```
                Function prototype:

   void WriteText(GuiWindow *gw, char *text, int x, int y)

Description:

   Writes text into a window in the current font and pen colour.

Parameters:

     gw: The guiwindow in which to write the text.
   text: The text to write.
      x: The x coordinate for the text.
      y: The y coordinate for the text.

Known bugs:

   None.

See also:


                 MakeOutputBox
```

1.190 GuiGetLastErr function

```
Function prototype:

   void GuiGetLastErr(char *error, char *file, int *line);

Description:

   Returns a description of the last internal Gui error to occur along with
   the file and line number where the error occurred.  Internal errors are
   usually caused by invalid parameters being sent to functions.

Parameters:

   error: A pointer to a string in which the error message will be
          returned.  This should be about 250 characters long in order to
          ensure that the error message will always fit.
    file: A pointer to a string in which the name of the file that the
          error occurred in will be returned.  This should be about 25
          characters long in order to ensure that the file name will fit,
          however, since gui errors can occur when calling FoxGUI macros
          (which are called from within your own programs) you should make
          sure that this string is at least as long as the name of the
          longest file name of your own source code + 1.
    line: A pointer to an integer in which the line number of the last
          error will be returned.  If FoxGUI does not report an error then
          this will be returned as 0.
```

Known bugs:

None.

1.191 GuiFree macro

Macro prototype:

```
void GuiFree(void *p)
```

Macro definition:

```
#define GuiFree(p)              GuiFreeMem(p,__LINE__,__FILE__)
```

Description:

This function is used to free memory allocated with the GuiMalloc function. It can work in one of two ways depending upon whether the function

UseSafeMallocs

has been called. If UseSafeMallocs has been called then GuiFree will detect any attempt to free memory that wasn't allocated by GuiMalloc or which has become corrupted due to some form of memory scribble. See the

GuiMalloc

function for more details.

Parameters:

p: A pointer to a section of memory allocated with the GuiMalloc function.

See also:

GuiMalloc

Warnings on the use of macros

1.192 SetDefaultCols function

Function prototype:

```
void SetDefaultCols(int BorderCol, int BackCol, int TextCol);
```

Description:

Sets the default colours used when creating FoxGUI controls. Calling this function before creating a control will ensure that these colors are used for that control and any others created subsequently. However,

```
   it will have no effect on any controls which have already been created
   which means that in theory you could have a different set of colours for
   every control.

Parameters:

   BorderCol: The colour to use when drawing borders.  This parameter is
              usually only used when creating a control if the THREED flag
              is not specified, otherwise a three-D border is created using
              the current Gui shine (high) and shadow (low) pens (see

                 SetGuiPens
                .
     BackCol: The colour to use for the background colour of controls.
              This will be ignored when the control is clear (e.g. if the
              BG_CLEAR flag is specified when creating a tick box or radio
              button).
     TextCol: The colour to use for any text in FoxGUI controls.

Known bugs:

   None.

See also:


                 SetGuiPens

                 SetDefaultFont
```

1.193 SetDefaultFont function

```
               Function prototype:

   void SetDefaultFont(char *name, int size, int style);

Description:

   Sets the default font used when creating FoxGUI controls.  Calling
   this function before creating a control will ensure that the specified
   font is used for that control and any others created subsequently.
   However, it will have no effect on any controls which have already been
   created which means that in theory you could have a different font for
   every control.

Parameters:

    name: The name of the font to use (e.g. "Topaz.font").
    size: The font size to use.
   style: The font style.  This is a combination of the following flags:
          FSF_UNDERLINED (if you want an underlined font), FSF_BOLD (if you
          want a bold font), FSF_ITALIC (if you want an italic font) and
          FSF_EXTENDED if you want an extra wide font.  You can combine any
          or all of these flags or you can pass 0 for a plain font.  These
```

```
          flags are not defined in FoxGUI.h but in the standard intuition
          header file graphics/text.h.

Known bugs:

   None.

See also:

                 SetDefaultCols
```

1.194 EnableM function

```
               Function prototype:

   void EnableM(int ObjectType, void *Parent, BOOL refresh);

Description:

   This function can be used to enable multiple objects.

Parameters:

   ObjectType: The type of object to enable.  This can be one of
               FrameTypeID, ButtonTypeID, TabControlTypeID, ListBoxTypeID,
               TreeControlTypeID, DDListBoxTypeID, EditBoxTypeID,
               OutputBoxTypeID, ProgressBarTypeID, TickBoxTypeID,
               RadioButtonTypeID, WindowTypeID, ScreenTypeID or
               TimerTypeID if you want to enable controls of one
               particular type or you can pass 0 to enable controls of all
               types.
       Parent: This should be a pointer to a GuiWindow if you only want to
               enable controls in a particular window or NULL if you want
               to enable all controls of the specified type.
      refresh: If TRUE then the controls are refreshed.


Known bugs:

   None.

Examples:

   EnableM(FrameTypeID, MyWindow, TRUE) will enable all frames in the
   window MyWindow and will refresh them.

   EnableM(0, MyWindow, FALSE) will enable all controls in window MyWindow
   but will not refresh them.

   EnableM(0, NULL, TRUE) will enable all controls and refresh them.

See also:
```

EnableControl

DisableM

1.195 GetDefaultFontCopy function

```
Function prototype:

   void GetDefaultFontCopy(char *fontname, int bufsize, int *height, int *style)

Description:

   This function gets the details of the current GUI font.

Parameters:

   fontname: A pointer to a character string to hold the returned
             fontname.
    bufsize: The size of the fontname buffer.
     height: Returns the height of the font.
      style: Returns the style of the font.

Known bugs:

   None.
```

1.196 DisableM function

```
               Function prototype:

   void DisableM(int ObjectType, void *Parent, BOOL refresh);

Description:

   This function can be used to disable multiple objects.

Parameters:

   ObjectType: The type of object to disable.  This can be one of
               FrameTypeID, ButtonTypeID, TabControlTypeID, ListBoxTypeID,
               TreeControlTypeID, DDListBoxTypeID, EditBoxTypeID,
               OutputBoxTypeID, ProgressBarTypeID, TickBoxTypeID,
               RadioButtonTypeID, WindowTypeID, ScreenTypeID or
               TimerTypeID if you want to disable controls of one
               particular type or you can pass 0 to disable controls of all
               types.
       Parent: This should be a pointer to a GuiWindow if you only want to
               disable controls in a particular window or NULL if you want
               to disable all controls of the specified type.
      refresh: If TRUE then the controls are refreshed.
```

```
Known bugs:

   None.

Examples:

   DisableM(FrameTypeID, MyWindow, TRUE) will disable all frames in the
   window MyWindow and will refresh them.

   DisableM(0, MyWindow, FALSE) will disable all controls in window MyWindow
   but will not refresh them.

   DisableM(0, NULL, TRUE) will disable all controls and refresh them.

See also:


                DisableControl

                EnableM
```

1.197 DestroyM function

```
               Function prototype:

   void DestroyM(int ObjectType, void *Parent, BOOL refresh);

Description:

   This function can be used to destroy multiple objects.

Parameters:

   ObjectType: The type of object to destroy.  This can be one of
               FrameTypeID, ButtonTypeID, TabControlTypeID, ListBoxTypeID,
               TreeControlTypeID, DDListBoxTypeID, EditBoxTypeID,
               OutputBoxTypeID, ProgressBarTypeID, TickBoxTypeID,
               RadioButtonTypeID, WindowTypeID, ScreenTypeID or
               TimerTypeID if you want to destroy controls of one
               particular type or you can pass 0 to destroy controls of all
               types.
       Parent: This should be a pointer to a GuiWindow if you only want to
               destroy controls in a particular window or NULL if you want
               to destroy all controls of the specified type.  This can
               also be a pointer to a FoxGUI screen if ObjectType is
               WindowTypeID.  In that case all windows in the specified
               screen will be closed.
      refresh: If TRUE then the controls are refreshed.


Known bugs:
```

```
   None.

Examples:

   DestroyM(FrameTypeID, MyWindow, TRUE) will destroy all frames in the
   window MyWindow and will refresh them.

   DestroyM(0, MyWindow, FALSE) will destroy all controls in window MyWindow
   but will not refresh them.

   DestroyM(0, NULL, TRUE) will destroy all controls (including windows and
   screens)  There will be nothing left to refresh!

See also:


                DisableM

                Destroy
```

1.198 UseSafeMallocs function

```
               Function prototype:

   void UseSafeMallocs(void);

Description:

   Tells FoxGUI to use safe memory allocations rather than fast memory
   allocations.  This modifies the function
                GuiMalloc
                to keep a record of
   allocations made and causes the function
                GuiFree
                to fail if an invalid
   pointer is passed.  However, it slows down these functions considerably
   so it is generally best to use safe mallocs while developing but then
   revert to fast mallocs when you are sure your code is bug free.  This
   function should be called directly after opening the FoxGUI library if
   you wish to use safe mallocs.

Known bugs:

   None.
```

1.199 Warnings and notes on the use of macros

```
   A small number of FoxGUI functions (listed below) are currently defined
   as macros rather than real functions.  This means that when you call the
   macro, the actual text of the macro gets substituted into your code by
```

the preprocessor rather than a link to the function being made at link time. Although you can use macros exactly as you would use functions there are a number of things you should bear in mind when doing so :-

Most compilers perform macro substitution on the first parse of your source code which means that if there is an error on the line with the macro on it and the compiler shows you the line with the error it won't look the way it did when you typed the line in and may make error spotting more difficult. To help you when this happens, definitions of all of the FoxGUI macros are included on their pages in this document.

It is not guaranteed that all FoxGUI macros will exist in their current form in later releases of the Gui. This doesn't mean that you shouldn't use them. There are two things that may cause a macro to change. The macro may be removed and implemented as a function or the underlying Gui structures may change necessitating a change in the macro definition. In either case the prototype will remain the same so there will be no need for you to change your code - you will just need to recompile it.

You should never be tempted to manually expand the macros. For example, the macro GetEditBoxID(p) is currently defined as (p)->id. Don't be tempted to use (p)->id in your functions. Always use GetEditBoxId(p). In this way you will ensure that your code will not have to be changed for future releases of FoxGUI where the macro implementation may be different.

The following FoxGUI functions are currently implemented as macros :-

@{ " GuiFree " Link GuiFree}

1.200 The GUI_END and GUI_CONTINUE flags

Most user defined callback functions (functions supplied by the ↩ user
that FoxGUI calls when certain events occur) have to return either GUI_END or GUI_CONTINUE. GUI_CONTINUE instructs FoxGUI to continue processing events. GUI_END causes event processing for this process to stop and the function
GuiLoop
to return.

1.201 The S_AUTO_SIZE flag

The S_AUTO_SIZE flag can be specified for all visible FoxGUI ↩ controls
except windows and screens. If any control has the S_AUTO_SIZE flag set and the window has a size gadget then resizing the window will cause the control to be resized and positioned so that it's size and position relative to the window remain the same.

See also:

```
OpenGuiWindow

MakeProgressBar

MakeFrame

MakeButton

MakeRadioButton

MakeTickBox

MakeEditBox

MakeListBox

MakeDDListBox

MakeOutputBox

MakeTabControlArray
```

1.202 Using FoxGUI with C++

```
Most of the documentation in this file assumes that you're using C
rather than C++ and that your compiler is operating in C mode even if it
is capable of compiling C++ programs.  However, it is possible to use
FoxGUI in a C++ program with some minor changes.

The first thing you have to do is define CPPSOURCE as follows before you
include FoxGUI.h in your code:

#define CPPSOURCE
#include "FoxGUI.h"

This modifies some of the prototypes of the FoxGUI functions to ensure
that no C++ name mangling occurs for the FoxGUI functions.

Any functions which are callbacks for FoxGUI functions should be defined
with using extern "C" e.g.

extern "C" int MyWinEventFn(GuiWindow *gw, int event, int x, int y, void * ↩
    DropData)
{
   // My code.
}

If you prefer, you can use EXTC which is defined in FoxGUI.h.
```

1.203 What's new in release 5.1?

```
                New features in release 5.1

*  Added the
                GetModeSize
                function to get the width and height of a display.
*  Added various flags to the
                GuiMessage
                function to support icons in the
   message.
*  Added the function
                WriteText
                which allows you to write text directly to
   a window without creating an outputbox.
*  Modified
                OpenGuiScreen
                to make the new screen public if PubName is not
   NULL.  It is no-longer necessary to pass the flag GS_PUBLIC.
*  Added function
                ClonePublicScreen
                to clone a public screen
   (requires V36).
*  Added function
                GetDefaultFontCopy
                which can be used to get a copy
   of a screens default font.
*  Added function
                GetScreenDetails
                to get the details of a public
   screen (requires V36).
*  Added S_FONT_SENSITIVE flag for output boxes and buttons.  Output
   box/button width/height is set according to font size and caption.
   Buttons with this flag set do not resize when in a resizable window
   (although they do still move).  Output boxes DO still resize because
   their contents may change and so resizing may still be important.
*  Changed the following macros into functions
                GetOutputBoxID
               ,
                SetPreText
               ,

                SetPostText
               ,
                WinPrint
               ,
                WinTab
               ,
                WinPrintTab
               ,
                WinPrintCol
               ,
                WinShowCursor
               ,

                WinHideCursor
```

```
                ,
                 WinClear
                ,
                 WinHome
                ,
                 WinBlankToEOL
                ,
                 WinWrapOn
                ,
                 WinWrapOff
                ,

                 GetEditBoxID
                ,
                 SetDDListBoxText
                ,
                 GetDDListBoxText
                ,
                 GetDDListBoxID
                .


Bug fixes in release 5.1

*  Fixed bug in
                 MakeFrame
                 which could cause a crash during drag/drop
   operations if you had not specified your own drag pointer.
*  Modified the function
                 MakeOutputBox
                 to take a void* instead of a
   Widget*.  This prevents you from having to cast parent objects as
   widgets when using strict compiler options or when compiling using C++.
```

1.204 What's new in release 5.0?

```
                New features in release 5.0

The main difference between release 4.7 and release 5.0 is that release 5.0
is implemented as a shared library whereas all previous releases were
implemented as a link library.  In order to make the change from a link
library to a shared library I had to reduce the number of parameters to
many of the FoxGUI functions.  This is because the easiest way to pass
parameters into a shared library is in registers and there are a limited
number of registers which can be used for this purpose.

Rather than simply remove parameters from each function until there were
enough registers to go around (which would have worked but would have made
FoxGUI even more inconsistent than it already was) I decided to aim for
some consistency.  This means that in some cases the number of parameters
to a function has decreased further than necessary,  simply to remain
consistent with other functions.  I hope that after the initial
inconvenience of making quite large changes to your old FoxGUI source code
you'll agree that this was a better approach.
```

The changes are listed in detail below.

* Functions which create controls and took a font as a parameter no-longer take a font parameter. Instead, the function
  SetDefaultFont
  can
  be used to set the default font to be used when creating controls. This works for all controls except edit boxes which always use the screen font (this is standard Amiga behaviour).
* Functions which create controls and took the border colour, the text colour and the background colour as parameters no-longer take these three parameters. Instead, the function
  SetDefaultCols
  can
  be used to set the default colours to be used when creating controls.
* Functions which create controls and took pre-text and post-text as parameters no-longer take these parameters. Instead, the functions

  SetPreText
  and
  SetPostText
  can be used.
* The functions MakeTickBox and MakeRadioButton no-longer take a caption parameter or accept the flags BG_CAPTION_LEFT or BG_CAPTION_RIGHT. Instead, the functions
  SetPreText
  and
  SetPostText
  can be used.
* The function GetEditBoxFloat has been replaced with the function

  GetEditBoxDouble
  .
* The function SetEditBoxFloat has been replaced with the function

  SetEditBoxDouble
  .
* The function SetOutputBoxFloat has been replaced with the function

  SetOutputBoxDouble
  .
* The function SetListBoxTabStops has been replaced with the function

  SetListBoxTabStopsArray
  .
* The functions
  MakeFrame
  ,
  MakeListBox
  and
  MakeTreeControl
  no-longer take
  a DragPointer as a parameter. The functions
  SetFrameDragPointer
  ,

```
                SetListBoxDragPointer
                and
                SetTreeControlDragPointer
                respectively can be
   used after creating your frame, list box or tree control if you want to
   specify your own drag pointer.  The flags LB_DRAGIMAGE, TC_DRAGIMAGE and
   FM_DRAGIMAGE are no-longer required and have been removed.
*  The function MakeTabControl has been replaced with the function

                MakeTabControlArray
               .
*  The function InitGui has been removed.  The initialisation that used to
   be performed in this function is now done by the library when you open
   it with the OpenLibrary command.  If you wish to use safe memory
   allocations (this used to be a flag which was passed to InitGui) you can
   now do so by calling the function
                UseSafeMallocs
                immediately after
   you open the FoxGUI library.  If you do not call UseSafeMallocs then
   fast memory allocations are used.
*  The function EndGui has been removed.  The cleaning up that used to be
   done by EndGui is now done when your application calls CloseLibrary to
   close the FoxGUI library.
*  The function
                OpenGuiWindow
                no-longer takes a pointer to a close
   function.  Instead close is handled by the event function along with the
   other events.
*  The HiCol parameter has been removed from the function
                OpenGuiWindow
               .
   Windows no-longer have a highlight colour and the macro WinPrintHi has
   also been removed.  Use
                WinPrintCol
                instead.
*  The colour parameters have been removed from the function

                ShowFileRequester
                which now uses the default Gui pen colours which can
   be set with the function
                SetDefaultCols
               .
*  The colour parameters have been removed from the function

                ShowDisplayList
                which now uses the default Gui pen colours which can
   be set with the function
                SetDefaultCols
               .
*  The colour parameters have been removed from the function

                GuiMessage
                which now uses the default Gui pen colours which can
   be set with the function
                SetDefaultCols
               .
*  The parameters MinWidth and MinHeight have been removed from the
```

```
   function
                OpenGuiWindow
               .  These are replaced by the more
   versatile function
                SetWindowLimits
               .
*  The function
                GuiTextLength
                now uses the default font if no font is
   specified.
*  Added the function
                EnableM
                to replace the functions EnableAllButtons,
   EnableWinButtons, EnableAllFrames, EnableWinFrames,
   EnableWinDDListBoxes, EnableAllDDListBoxes, EnableWinEditBoxes,
   EnableAllEditBoxes, EnableAllListBoxes, EnableWinListBoxes and
   EnableEverything which have been removed.
*  Added the function
                DisableM
                to replace the functions DisableAllButtons,
   DisableWinButtons, DisableAllFrames, DisableWinFrames,
   DisableWinDDListBoxes, DisableAllDDListBoxes, DisableWinEditBoxes,
   DisableAllEditBoxes, DisableAllListBoxes, DisableWinListBoxes and
   DisableEverything which have been removed.
*  Added the function
                DestroyM
                to replace the functions DestroyAllButtons,
   DestroyWinButtons, DestroyWinTabControls,DestroyAllTabControls,
   DestroyAllFrames, DestroyWinFrames,DestroyAllRadioButtons,
   DestroyWinRadioButtons, DestroyWinTickBoxes,DestroyAllTickBoxes,
   DestroyAllDDListBoxes, DestroyWinDDListBoxes,DestroyAllOutputBoxes,
   DestroyWinOutputBoxes, DestroyAllListBoxes,DestroyWinListBoxes,
   DestroyAllEditBoxes, DestroyWinEditBoxes, DestroyAllTimers,
   CloseScrWindows, CloseAllWindows and CloseAllGuiScreens which have been
   removed.
*  A new parameter InitialValue has been added to the function

                MakeOutputBox
                which sets the initial value of the text
   in the output box.
*  The functions DestroyButton, DestroyTabControl, DestroyFrame,
   DestroyRadioButton, DestroyTickBox, DestroyDDListBox, DestroyEditBox,
   DestroyOutputBox, DestroyListBox, DestroyProgressBar, DestroyTimer,
   CloseGuiWindow and CloseGuiScreen have been removed.  Use the function

                Destroy
                instead.
*  The functions StoreDDListBoxStatus and RestoreDDListBoxStatus have been
   removed.
*  The functions StoreEveryStatus and RestoreEveryStatus have been
   removed.
*  The functions StoreEditBoxStatus and RestoreEditBoxStatus have been
   removed.
*  The functions StoreButtonStatus and RestoreButtonStatus have been
   removed.
*  The functions EnableRadioButton, EnableTickBox, EnableFrame,
   EnableButton, EnableDDListBox, EnableEditBox, EnableListBox and
```

```
   EnableTabControl have been removed.  Use the function
                EnableControl
                  instead.
*  The functions DisableRadioButton, DisableTickBox, DisableFrame,
   DisableButton, DisableDDListBox, DisableEditBox, DisableListBox and
   DisableTabControl have been removed.  Use the function
                DisableControl
                  instead.


Bug fixes in release 5.0

*  Fixed bug which caused GuiMessage to crash if the message was an empty
   string.
```

1.205 What's new in release 4.7?

```
                New features in release 4.7

*  New flags GW_DISKIN and GW_DISKOUT can be passed to the
                OpenGuiWindow
                  function if you want your window to respond to disks being  ↩
                     inserted and
   removed.
*  The function ReplaceTCItem has been modified so that it can be passed
   the return value from the function TCItemText on the same item without
   crashing.  This gives you an easy way to keep the text of the item the
   same but with a different image.

Bug fixes in release 4.7

*  Fixed bug which caused memory scribble when ReplaceTCItem was called for
   an item not currently visible due to it's parent folder being closed.
*  Fixed many bugs in RemoveItem.
```

1.206 What's new in release 4.6?

```
                New features in release 4.6

*  The functions
                GuiTextLength
                and
                ReplaceTCItem
               *  The flags TC_REHILIGHT_ON_SCROLL and LB_REHILIGHT_ON_SCROLL to  ↩
                  modify
   the way list boxes and tree controls rehilight when scrolling them.

Bug fixes in release 4.6

None.
```

1.207 What's new in release 4.5?

```
New features in release 4.5

*  Tree Control

Bug fixes in release 4.5

*  Fixed list box bug which made it impossible to select items at the
   bottom of the list box if the list box had titles.
*  Fixed bug which caused dragging to start if you double clicked on an
   item in a list box which had been opened with the LB_DBLCLICK and
   LB_DRAG flags.
*  Fixed list box bug which caused a single click on an item to be treated
   as a double-click if it immediately followed a successful double-click.
```

1.208 What's new in release 4.4?

```
New features in release 4.4

*  Frames can now be enabled and disabled using the functions EnableFrame,
   DisableFrame, EnableAllFrames, DisableAllFrames, EnableWinFrames,
   DisableWinFrames, EnableControl and DisableControl.
*  New flags FM_BORDERLESS and FM_DRAGOUTLINE added for frames, allowing
   the creation of borderless frames and frames which show their outline as
   they are dragged.
*  GetModeName function added to get the name of a screen mode from the
   display database.
*  It is now possible to create a tab control with only one tab (though why
   you would want to do so I'm not sure!)  Previously the function failed
   unless at least two were specified.
*  SetTickBoxValue function added.
*  The look of list boxes with titles has been dramatically improved.  The
   titles are now drawn in a separate bevel above the bevel containing the
   items.
*  Tabbed listboxes have been fixed so that text in a column is truncated
   before it encroaches on the space of the next column.  Unfortunately
   this has caused a speed reduction when scrolling tabbed listboxes.
*  InsertListBoxItem function added to allow better manipulation of items
   in listboxes.
*  It is now possible to unhilight a listbox by passing 0 as the item
   number to SetListBoxHiNum.

*  The NoLines function has been modified to fit in with the new look for
   listboxes.  Previously it returned the number of lines available in a
   listbox.  Those lines could be used for titles or items.  Now it returns
   the space available for showing items.

Bug fixes in release 4.4
```

* Fixed bug in MakeFrame which could incorrectly positioned frame captions.
* Fixed bug which could cause a crash if a windows event function destroyed the source window when a frame was dropped in it.
* Fixed bug which stopped bitmaps from being attached to frames if the frame wasn't created with the FM_LBUT flag.
* Fixed bug which caused the whole image to be shown if a button or frame containing a user-clipped image was resized.
* Fixed bug which caused vertical scrollers on listboxes to be created before they were necessary.
* Fixed crash when creating a radio button with no caption.

Errors in this documentation which have been corrected.

* The documentation for the function ListColumnText incorrectly stated that the leftmost column is column 1 - it is 0.
* The documentation for the function SetListBoxHiNum contained an error in the example code.

1.209 What's new in release 4.3?

Important note for users of FoxGUI 4.2

If you have been using FoxGUI 4.2, when you upgrade to 4.3 you will need to make some changes to your code. The functions
ShowFileRequester
and
OpenGuiWindow
have had major revisions and your code will need to be modifed accordingly.

New features in release 4.3

* A Cursor event has been added for list boxes. You can now specify a function to be called when the up and down cursor keys are used to select an item. See
MakeListBox
.
* Added list box functions to find items (
FindListText
), individually replace items (
ReplaceListBoxItem
), get the text of a specified column (
ListColumnText
) and set the hilighted element (
SetListBoxHiElem
).
* If a list box is tabbed, the tabs now move if the list box resizes.
* It is now possible to create backdrop windows with
OpenGuiWindow
.
* Callback functions can now be called when a GuiWindow is moved, resized or activated by the user (see

```
                 OpenGuiWindow
                ).
*  Added functions to modify and enquire about the state of check-mark menu
   items (
                 IsMenuChecked
                 and
                 SetMenuChecked
                ).
*  Added C++ compatability (see
                 Using FoxGUI with C++
                ).
*  Reduced the number of parameters required by the
                 ShowFileRequester
                   function.
*  Added functions for handling display modes and creating lists of
   available display modes (
                 GetNextAvailableDisplayMode
                 and

                 ShowDisplayList
                ).

Bug fixes in release 4.3

*  Fixed default font selection for list boxes.  If no font is specified,
   the list box will inherit the font used by the parent screen.
*  Fixed bug in Destroy() which could cause crashes when destroying
   GuiWindows.
```

1.210 What's new in release 4.2?

```
New features in release 4.2

*  Drag/Drop functionality has been added to windows, frames and listboxes.
   To accomodate these changes, various function prototypes have changed
   slightly including the functions MakeFrame, MakeListBox and
   OpenGuiWindow.
*  A double click-event is now available for list boxes.  You can now
   specify a function to call if the user double-clicks on an item in a
   list box control.
*  Other improvements to list boxes: They automatically get a vertical
   scroller when more items are added than can be shown in the list and
   they automatically get a horizontal scroller when items are added which
   are too long to be shown in full.

Bug fixes in release 4.2

*  If a frame was created without specifying the FM_LBUT flag then right
   mouse-button clicks on the frame would not be detected and the frame's
   title (if it had one) would appear above and to the left of the frame
   instead of in the centre.  This is now fixed.
```

1.211 What's new in release 4.1?

```
New features in release 4.1

*  Added the function SetGuiPensFromPubScreen which sets the gui pens to
   the values used by a specified public screen.
*  Tab controls now have rounded corners.

Bug fixes in release 4.1

* The "far" directive in foxconsole.h has been replaced with "__far" for
  ANSI compliance.
* A bug in the way that drop-down list boxes were destroyed and which could
  have caused the machine to crash if the list box had a child or if the
  list box was associated has been fixed.
* The descriptions of the parameters detail and Btext were incorrect for
  the function GuiMessage as described in this documentation.  This has
  been corrected.
```

1.212 What's new in release 4.0?

```
New features in release 4.0

*  Recompiled all of the code with SAS/C 6.51.  The code now requires
   sc.lib and scm.lib to be linked with it rather than the old lc.lib and
   lcm.lib which weren't compatible with newer versions of the compiler.
   This should allow far more people to make use of FoxGUI.
*  The ability to add your own intuition gadgets to FoxGUI windows or to
   your own windows within a FoxGUI program.

Bug fixes in release 4.0

*  If the cursor keys were used to scroll list boxes then hidden scrollable
   list boxes would also be redrawn.  This is now fixed.
*  If two listboxes in different frames of the same tab control overlapped
   then it was impossible to select items in one of the list boxes by
   clicking in the overlapping region.  This is now fixed.
*  It is no-longer possible for buttons in GuiMessage windows to overlap.
```

1.213 What's new in release 3.0?

```
New features in release 3.0

*  Tab controls.
*  Timer controls.
*  All controls can be hidden and shown.
*  Custom borders in buttons and frames are now resized when the button or
   frame resizes.
*  Frames and tab controls can now be used as holders for other controls.
*  The parameter list for the function SetOutputBoxCols has changed
   slightly.
*  The way that list boxes work has been slightly improved.
```

```
* New flags have been added to create clear edit boxes and drop-down list
  boxes.
* A whole new set of generic functions which work on any control type have
  been added.  The functions are:  Hide, Show, DisableControl,
  EnableControl, Destroy and GetWindow.
* A new function RemoveMenuItem has been added.
* Enhancements have been made to the GuiMessage function.

Bug fixes in release 3.0

* The function TickBoxValue always returned FALSE no matter what the
  actual value was.  This has been fixed.
```

1.214 What's new in release 2.0?

```
New features in release 2.0

* Much better screen support (Interlace, new screen modes etc).
* Windows can be resized.
* All controls are capable of stretching, shrinking and moving to fit the
  new bounds of resized windows.
* ILBM images can be loaded, drawn on windows and attached to buttons and
  frames (either clipped or scaled to fit exactly).
* Tick boxes can be clear.
* Pop-up list boxes and sub drop-down list boxes can be made to calculate
  the necessary width for you so that you don't have to work it out
  yourself.  This is useful if you don't know what font will be in use.

Bug fixes in release 2.0

* Text in frames, buttons and output boxes is now clipped to fit.
  Previously it would often overflow out of the control.  If the control
  is resized (due to a window resizing) then the caption is reclipped.
* On an A500, if a long item was selected in a drop-down list box,
  selecting another, shorter item would sometimes leave the last
  character of the previous text behind.  This is now fixed.
* If you don't want a caption in a button or frame you can now pass NULL
  for the name parameter.  Previously you had to use an empty string ("").
* The SetEditBoxFocus() function used to fail if another string gadget
  (i.e. another edit box or drop-down list box) was active.  This is now
  fixed.
* There were various bugs associated with font handling which are now
  fixed.
```

1.215 ... macro

```
                Macro prototype:

Macro definition:

Description:
```

```
Parameters:

Returns:

See also:

                Warnings on the use of macros
```

1.216 ... function

```
Function prototype:

Description:

Parameters:

Returns:

Known bugs:

   None.

See also:
```