MUI - MagicUserInterface

A system to create and maintain graphical user interfaces

- Programmer Documentation -

Version 2.2
08-Aug-1994

Stefan Stuntz

1 Getting Started

Note: This documentation does not cover all concepts of MUI programming in detail. It's important that you also read the accompanying per class autodocs and have a look at the supplied demo programs!

1.1 Object Oriented Programming

The MUI system is based on BOOPSI, the Basic Object Oriented Programming System for Intuition. Understanding MUI and understanding this documentation requires at least a little knowledge about the concepts of object oriented programming, about classes, objects, methods and attributes. An absolutely sufficient introduction to these topics can be found in the "Libraries" part of the "ROM Kernel Reference Manuals" or in several Amiga mail articles.

When talking about BOOPSI, most people automatically think of BOOPSI images and BOOPSI gadgets as part of the Amiga operating system. However, BOOPSI for itself is just a system for object oriented programming. One could e.g. have object oriented spread sheet software or object oriented file systems based on BOOPSI, intuition's builtin classes (gadgetclass, imageclass) are just two from thousands of possibilities.

The MUI system also uses BOOPSI only as a base for object oriented programming. Thus, MUI classes are all subclasses of BOOPSI's rootclass and have nothing in common with the system supplied gadget or image classes. Unfortunately, Commodore missed some very important topics when designing these classes, disabling them for use in automatic layout systems such as MUI. Anyway, MUI features an interface to BOOPSI's gadgetclass and allows using already available gadgets (e.g. the Kick 3.x colorwheel) in MUI applications.

1.2 Available Classes

The MUI system comes with several classes, each of them available as seperate shared system library. These classes are organized in a tree. As usual in the OO programming model, objects inherit all methods and attributes from their true class as well as from all their superclasses. Here is a quick summary with some short notes what the classes are used for. More detailed information can be found later in this document and in the per class autodocs files coming with the developer archive.

```
rootclass               (BOOPSI's base class)
\--Notify               (implements notification mechanism)
   +--Application       (main class for all applications)
   +--Window            (handles intuition window related topics)
   \--Area              (base class for all GUI elements)
      +--Rectangle      (creates empty rectangles)
      +--Image          (creates images)
      +--Text           (creates some text)
      +--String         (creates a string gadget)
      +--Prop           (creates a proportional gadget)
      +--Gauge          (creates a fule gauge)
      +--Scale          (creates a percentage scale)
      +--Boopsi         (interface to BOOPSI gadgets)
      +--Colorfield     (creates a field with changeable color)
      +--List           (creates a line-oriented list)
      !  +--Floattext   (special list with floating text)
      !  +--Volumelist  (special list with volumes)
      !  +--Scrmodelist (special list with screen modes)
      !  \--Dirlist     (special list with files)
      \--Group          (groups other GUI elements - handles layout)
         +--Virtgroup   (handles virtual groups)
         +--Scrollgroup (handles virtual groups with scrollers)
         +--Scrollbar   (creates a scrollbar)
         +--Listview    (creates a listview)
         +--Radio       (creates radio buttons)
         +--Cycle       (creates cycle gadgets)
         +--Slider      (creates slider gadgets)
         +--Coloradjust (creates some RGB sliders)
         \--Palette     (creates a complete palette gadget)
```

1.3 Application Theory

A MUI application consists of a (sometimes very) big object tree (don't mix up with the class tree explained above). The root of this tree is always an instance of application class, called application object. This application object handles the various communication channels such as user input through windows, ARexx commands or commodities messages.

An application object itself would be enough to create non-GUI programs with just ARexx and commodities capabilities. If you want to have windows with lots of nice gadgets and other user interface stuff, you will have to add window objects to your application. Since the application object is able to to handle any number of children, the number of windows is not limited.

Window objects are instances of window class and handle all the actions related with opening, closing, moving, resizing and refreshing of intuition windows. However, a window for itself is not

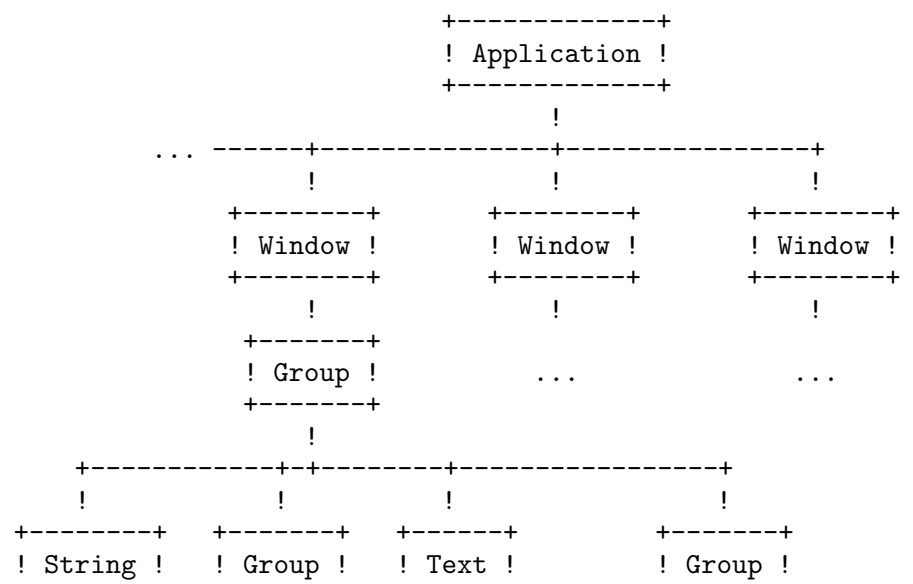
of much use without having any contents to display. That's why window objects always need a so called root object.

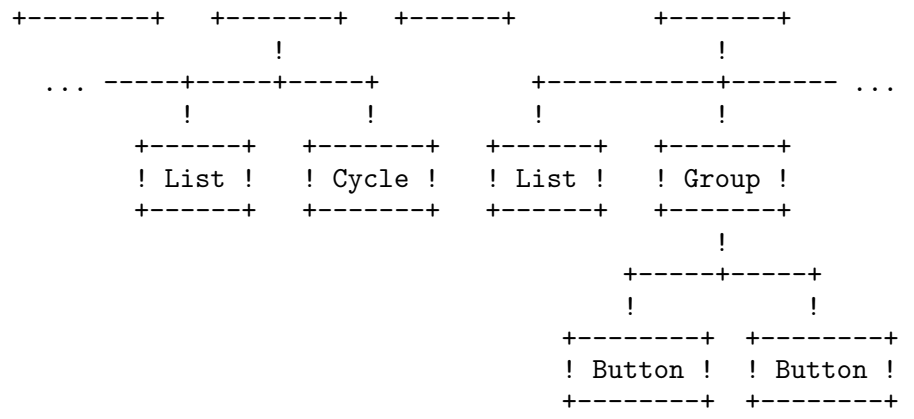
With this root object, we finally reach the gadget related classes of the MUI system. These gadget related classes are all subclasses of area class, they describe a rectangle region with some class dependant contents. Many different classes such as strings, buttons, checkmarks or listviews are available, but the most important subclass of area class is probably the group class. Instances of this class are able to handle any number of child objects and control the size and position of these children with various attributes. Of course these children can again be group objects with other sets of children. Since you usually want your window to contain more than just one object, the root object of a window will be a group class object in almost all cases.

Because these first paragraphs are very important to understand how MUI works, here's a brief summary:

An application consists of exactly one application object. This application object may have any number of children, each of them being a window object. Every window object contains a root object, usually of type group class. This group object again handles any number of child objects, either other group objects or some user interface elements such as strings, sliders or buttons.

A little diagram might make things more clear:





As shown in this tree, only three types of objects are allowed to have children:

```
Application: zero or more children of window class.
     Window: exactly one child of any subclass of area class.
      Group: one or more children of any subclass of area class.
```

1.4 Object Handling

Since MUI uses BOOPSI as object oriented programming system, objects could simply be created using intuition.library/NewObject(). However, '`muimaster.library`' also features a generation function called

```
Object * MUI_NewObjectA(STRPTR class, struct TagItem *taglist);
```

with the varargs stub

```
Object * MUI_NewObject(STRPTR class, Tag tag1, ..., TAG_DONE);
```

That's the function you should use when creating objects of public MUI classes. The parameter '`class`' specifies the name of the object's class (e.g. '`MUIC_Window`', '`MUIC_Slider`', ...). If the needed class isn't already in memory, it is automatically loaded from disk.

With '`taglist`', you specify initial create time attributes for your object. Every attribute from the objects true class or from one of its super classes is valid here, as long as it's marked with the letter '`I`' in the accompanying autodocs documentation.

To create a string object with a string-kind frame, a maximum length of 80 and the initial contents "foobar", you would have to use the following command:

```
MyString = MUI_NewObject(MUIC_String,
               MUIA_Frame          , MUIV_Frame_String,
               MUIA_String_Contents, "foobar",
               MUIA_String_MaxLen  , 80,
               TAG_DONE);
```

Once your object is ready, you can start talking to it by setting or getting one of its attributes or by sending it methods. The standard BOOPSI functions '`SetAttrs()`', '`GetAttr()`' and '`DoMethod()`' are used for these purposes:

```
~  char *contents;
~  SetAttrs(MyString,MUIA_String_Contents,"look",TAG_DONE);
~  GetAttr(MUIA_String_Contents,MyString,&contents);
   printf("Always %s on the bright side of life.",contents);

   DoMethod(mylist,MUIM_List_Remove,42); /* remove entry nr 42 */
```

As already mentioned above, all attributes and methods are completely documented in the autodocs coming with this distribution. These autodocs follow the usual format, you can parse them with one of the various tools to create some hypertext online help for your favourite editor.

When you're done with an object, you should delete it with a call to

```
VOID MUI_DisposeObject(Object *obj);
```

from '`muimaster.library`'. After doing so, the object pointer is invalid and must no longer be used.

When deleting objects, the parent-child connections mentioned above play an important role. If you dispose an object with children, not only the object itself but also all of its children (and their children, and the children of their children ...) get deleted. Since in a usual MUI application, the application object is the father of every window, the window is the father of it's contents and every group is the father of its sub objects, a single dispose of the application object will free the entire application.

Note well: you may **not** delete objects that are currently children of other objects. Thus, if you have a complete application tree, the only thing you can delete is the application object itself as

this one has no father. You can, however, add and remove children dynamically. More information on that topic follows later in this document.

1.5 Macros

This chapter is only valid if you use C as your MUI programming language. Other language interfaces might feature other types of macros or support functions. Please have a look at the supplied interfaces to see how they work.

The tree structure that builds up an application also appears in the source code of a MUI program. Since adding child objects is always possible with a special attribute of the parent object, it is common to create the whole tree with one big function call.

To help making these calls more clear, the MUI header files contain several macros that simplify the task of object generation.

Instead of

```
MUI_NewObject(MUIC_Window, ..., TAG_DONE);
MUI_NewObject(MUIC_String, ..., TAG_DONE);
MUI_NewObject(MUIC_Slider, ..., TAG_DONE);
```

you can simply use

```
WindowObject, ..., End;
StringObject, ..., End;
SliderObject, ..., End;
```

Please note that the '`xxxObject`' macros contain an opening bracket and thus must always be terminated with an '`End`' macro that contains the matching closing bracket.

Besides these "two way" macros, there are also some complete object definitions available which all create specific objects with certain types of attributes. The macro

```
SimpleButton("Cancel")
```

would e.g. generate a complete button object with the correct frame, background and input capabilities. Though lots of these types of macros are available and can of course be used directly in

your applications, they are mainly intended as some kind of example. Usually you will need some more sophisticated generation capabilities with a more specific set of macros.

Note: If your application needs lots of objects from a specific type (e.g. 200 buttons), you can save some memory by turning macros into functions.

2 Layout Engine

2.1 Overview

One of the most important and powerful features of MUI is its dynamic layout engine. As opposed to other available user interface tools, the programmer of a MUI application doesn't have to care about gadget sizes and positions. MUI handles all necessary calculations automatically, making every program completely screen, window size and font sensitive without the need for the slightest programmer interaction.

From a programmers point of view, all you have to do is to define some rectangle areas that shall contain the objects you want to see in your window. Objects of group class are used for this purpose. These objects are not visible themselves, but instead tell their children whether they should appear horizontally or vertically (there are more sophisticated layout possibilities, more on this later).

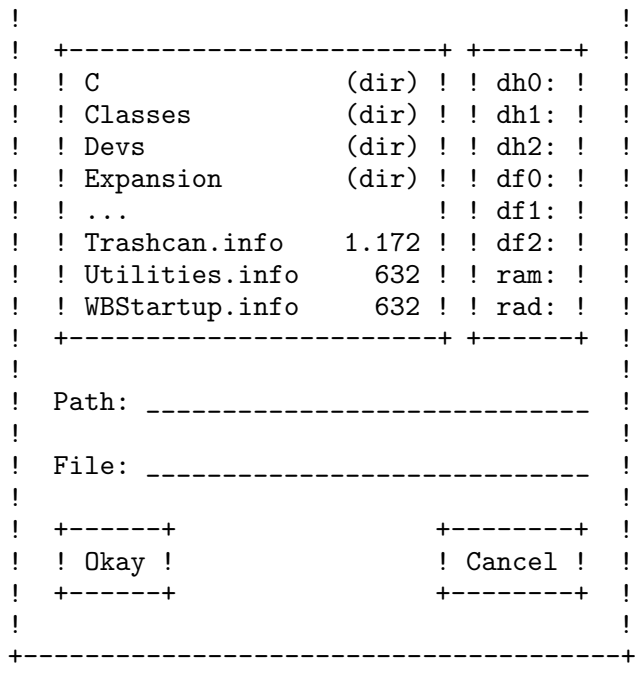
For automatic and dynamic layout, it's important that every single object knows about its minimum and maximum dimensions. Before opening a window, MUI asks all its gadgets about these values and uses them to calculate the windows extreme sizes.

Once the window is opened, layout takes place. Starting with the current window size, the root object and all its children are placed depending on the type of their father's group and on some additional attributes. The algorithm ensures that objects will never become smaller as their minimum or larger as their maximum size.

The important thing with this mechanism is that object placement depends on window size. This allows very easy implementation of a sizing gadget: whenever the user resizes a window, MUI simply starts a new layout process and recalculates object positions and sizes automatically. No programmer interaction is needed.

2.2 Groups

As mentioned above, a programmer specifies a windows design by grouping objects either horizontally or vertically. As a little example, lets have a look at a simple file requester window:



This window consists of two listview objects, two string gadgets and two buttons. To tell MUI how these objects shall be placed, you need to define groups around them. Here, the window consists of a vertical group that contains a horizontal group with both lists as first child, the path gadget as second child, the file gadget as third child and again a horizontal group with both buttons as fourth child.

Using the previously defined macro language, the specification could look like this (in this example, '`VGroup`' creates a vertical group and '`HGroup`' creates a horizontal group):

```
VGroup,
   Child, HGroup,
      Child, FileListview(),
      Child, DeviceListview(),
      End,
   Child, PathGadget(),
   Child, FileGadget(),
   Child, HGroup,
      Child, OkayButton(),
      Child, CancelButton(),
      End,
   End;
```

This tiny piece of source is completely enough to define the contents of our window, all necessary sizes and positions are automatically calculated by the MUI system.

To understand how these calculations work, it's important to know that all basic objects (e.g. strings, buttons, lists) have a fixed minimum and a maximum size. Group objects calculate their minimum and maximum sizes from their children, depending whether they are horizontal or vertical:

- Horizontal groups
  The minimum width of a horizontal group is the sum of all minimum widths of its children.
  The maximum width of a horizontal group is the sum of all maximum widths of its children.
  The minimum height of a horizontal group is the biggest minimum height of its children.
  The maximum height of a horizontal group is the smallest maximum height of its children.
- Vertical groups
  The minimum height of a vertical group is the sum of all minimum heights of its children.
  The maximum height of a vertical group is the sum of all maximum heights of its children.
  The minimum width of a vertical group is the biggest minimum width of its children.
  The maximum width of a vertical group is the smallest maximum width of its children.

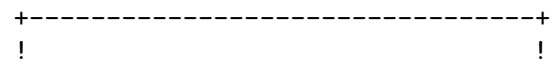
Maybe this algorithm sounds a little complicated, but in fact it is really straight forward and ensures that objects will neither get smaller as their minimum nor bigger as their maximum size.

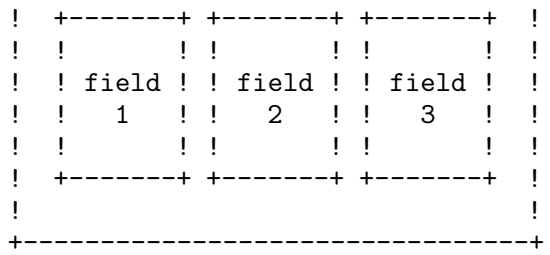
Before a window is opened, it asks its root object (usually a group object) to calculate minimum and maximum sizes. These sizes are used as the windows bounding dimensions, the smallest possible window size will result in all objects being display in their minimum size.

Once minimum and maximum sizes are calculated, layout process starts. The root object is told to place itself in the rectangle defined by the current window size. This window size is either specified by the programmer or results from a window resize operation by the user. When an object is told to layout itself, it simply sets its position and dimensions to the given rectangle. In case of a group object, a more or less complicated algorithm distributes all available space between its children and tells them to layout too.

This "more or less complicated algorithm" is responsible for the object arrangement. Depending on some attributes of the group object (horizontal or vertical, ...) and on some attributes of the children (minimum and maximum dimensions, ...), space is distributed and children are placed.

A little example makes things more clear. Let's see what happens in a window that contains nothing but three horizontally grouped colorfield objects:





Colorfield objects have a minimum width and height of one pixel and no (in fact a very big) maximum width and height. Since we have a horizontal group, the minmax calculation explained above yields to a minimum width of three pixels and a minimum height of one pixel for the windows root object (the horizontal group containing the colorfields). Maximum dimensions of the group are unlimited. Using these results, MUI is able to calculate the windows bounding dimensions by adding some spacing values and window border thicknesses.

Once min and max dimensions are calculated, the window can be opened with a programmer or user specified size. This size is the starting point for the following layout calculations. For our little example, let's imagine that the current window size is 100 pixels wide and 50 pixels high.

MUI subtracts the window borders and some window inner spacing and tells the root object to layout itself into the rectangle left=5, top=20, width=90, height=74. Since our root object is a horizontal group in this case, it knows that each colorfield can get the full height of 74 pixels and that the available width of 90 pixels needs to be shared by all three fields. Thus, the resulting fields will all get a width of 90/3=30 pixels.

That's the basic way MUI's layout system works. There are a lot more possibilities to influence layout, you can e.g. assign different weights to objects, define some inter object spacing or even make two-dimensional groups. These sophisticated layout issues are discussed in the autodocs of group class.

3 Building An Application

3.1 Creation

Creating all the objects that make up an applications user interface is usually done with one big '`MUI_NewObject()`' call. This call returns a pointer to the application object as its result and contains lots of other object creation calls as parameter for its tag items. Using the previously defined macro language, a sample generation call could look like this:

```
app = ApplicationObject,
   MUIA_Application_Title      , "Settings",
   MUIA_Application_Version    , "$VER: Settings 6.16 (20.10.93)",
   MUIA_Application_Copyright  , "©1992/93, Stefan Stuntz",
   MUIA_Application_Author     , "Stefan Stuntz",
   MUIA_Application_Description, "Just a silly demo",
   MUIA_Application_Base       , "SETTINGS",

   SubWindow, window1 = WindowObject,
      MUIA_Window_Title, "Save/use me and start me again!",
      MUIA_Window_ID   , MAKE_ID('S','E','T','T'),

      WindowContents, VGroup,

         Child, ColGroup(2), GroupFrameT("User Identification"),
            Child, Label2("Name:"  ), Child, str1 = String(0,40),
            Child, Label2("Street:"), Child, str2 = String(0,40),
            Child, Label2("City:"  ), Child, str3 = String(0,40),
            Child, Label1("Passwd:"), Child, str4 = String(0,40),
            Child, Label1("Sex:"   ), Child, str5 = String(0,40),
            Child, Label("Age:"),
            Child, sl  = SliderObject, End,
            End,

         Child, VSpace(2),

         Child, HGroup,
            MUIA_Group_SameSize, TRUE,
            Child, btsave   = KeyButton("Save"  ,'s'),
            Child, btuse    = KeyButton("Use"   ,'u'),
            Child, btcancel = KeyButton("Cancel",'>'),
            End,

         End,
      End,
```

```
      SubWindow, window2 = WindowObject,
         MUIA_Window_Title, "Window 2",
         ...,
         ...,
         End,

      SubWindow,
         ...
         End,

      End;

   if (!app) fail(app,"Failed to create Application.");
```

This big structure is indeed one single function call that builds a lot of other objects on the fly. Windows are created as children of the application object, a windows contents are created as child of the window and a groups contents are created as children of the group.

Though many single objects are created, error handling is very easy. When a parent object encounters a NULL pointer supplied as one of its children, it will automatically dispose all other supplied children and fail too. Thus, even errors occuring in a very deep level will cause the complete application object to fail. On the other hand, if you receive a non NULL application pointer, you can be sure that all other objects have successfully been created.

Once you're done with your application, a single

```
MUI_DisposeObject(app);
```

is enough to get rid of all previously created objects.

3.2 Notificiation

The central element for controlling a MUI application is the notification mechanism. To understand how it works, its important to know that most objects feature lots of attributes that define their current state. Notification makes it possible to react on changes of these attributes.

Attributes are changed either directly by the programmer (with a call to SetAttrs()) or by the user manipulation some gadgets. If he e.g. drags around the knob of a proportional gadget, the MUIA_Prop_First attribute will continously be updated and reflect the current position.

With notification, you could directly use this attribute change to set the MUIA_List_TopPixel attribute of a list object, building up a full featured listview:

```
DoMethod(sbar, MUIM_Notify, MUIA_Prop_First, MUIV_EveryTime,
         list, 3, MUIM_Set, MUIA_List_TopPixel, MUIV_TriggerValue);
```

To make it clear: Every time, the scrollbar object changes its '`MUIA_Prop_First`' value, the list object shall change its '`MUIA_List_TopPixel`' attribute accordingly. The value 3 in the above function call identifies the number of the following parameters. Since you can call any method with any parameters here and MUI needs to save it somewhere, it's important to set it correctly.

From now on the list and the scrollbar are connected to each other. As soon as the proportional gadget is moved, the position of the list changes accordingly; the programmer doesn't have to care about.

Notification is mostly used together with either the '`MUIM_Application_ReturnID`' or with the '`MUIM_CallHook`' method. If you e.g. have a specific hook that should be called whenever the user presses a button, you could use the following notify method:

```
DoMethod(button, MUIM_Notify, MUIA_Button_Pressed, FALSE,
         button, 2, MUIM_CallHook, &ButtonHook);

/* Whenever the button's pressed attribute is set to FALSE
   (i.e. the user has released the button), the button object
   itself will call ButtonHook. */
```

Futher information can be found in the autodocs of notify class.

4 Dynamic Object Linking

4.1 Overview

Usually, the complete user interface of an application is created with one single command. This makes error handling very easy and allows parallel usage of several windows. However, sometimes it makes sense to create certain windows only when they are actually needed: For example, if an application supplies many subwindows and would use too much memory, or if the number and contents of needed windows is not known at application startup time.

Therefore MUI supports the option of "late binding". Using this mechanism, children can be added and removed after their parent object already has been created. MUI uses the methods '`OM_ADDMEMBER`' and '`OM_REMMEMBER`' for this purpose:

```
DoMethod(parent,OM_ADDMEMBER,child); /* add child object    */
DoMethod(parent,OM_REMMEMBER,child); /* remove child object */
```

Both methods are only supported by MUI's application and group class; these are the only classes that can manage several children. Dynamic object linking for window and group class is explained in detail in the following chapters.

Note: Objects that do not have parents, be it, because they are not yet connected using '`OM_ADDMEMBER`' or because they were disconnected using '`OM_REMMEMBER`', it's the programmer's task to delete them by calling '`MUI_DisposeObject()`'. On the other side, objects that still are children of other objects must not be deleted!

4.2 Dynamic Windows

Let's say an application object is already set up and another (not yet existing) window has to be added. First, the window object needs to be created:

```
win = WindowObject,
   MUIA_Window_Title, "New Window",
      WindowContents, VGroup,
         Child, ...,
         Child, ...,
         Child, ...,
         End,
```

```
~     End,
   End;

if (!win) fail(); /* failure check */
```

After the window object is created, it can be added to the application as one of its children:

```
DoMethod(app,OM_ADDMEMBER,win);
```

Now this window has become a part of the application, just as if it had been created as a subwindow together with the application object. It can be opened and closed by setting the according attributes and will be deleted automatically as soon as the application is ended.

Usually, however, you'll want to delete this window directly after usage, because the late binding wouldn't make much sense otherwise.

After closing the window via

```
set(win,MUIA_Window_Open,FALSE);
```

you can remove it by calling

```
DoMethod(app,OM_REMMEMBER,win);
```

After this you have to delete the window object "by hand", since the application no longer knows of it:

```
MUI_DisposeObject(win);
```

This method makes it possible to create subroutines that open their own window, wait for some imput events und return something.

To illustrate this, here is a short example:

```
set(app,MUIA_Application_Sleep,TRUE);  // disable other windows

win = WindowObject, ...., End;         // create new window

if (win)                               // ok ?
```

```
{
    DoMethod(app,OM_ADDMEMBER,win);    // add window...
    set(win,MUIA_Window_Open,TRUE);    // and open it

    while (running)
    {
        switch (DoMethod(app,MUIM_Application_Input,&sigs))
        {
            ... // Extra Input loop. For this window only.
            ... // Note: The special value
            ... // MUIV_Application_ReturnID_Quit should be recognized
            ... // as well
        }
    }

    set(win,MUIA_Window_Open,FALSE);   // Close window

    DoMethod(app,OM_REMMEMBER,win);    // remove

    MUI_DisposeObject(win);            // and kill it
}

set(app,MUIA_Application_Sleep,FALSE); // wake up the application
```

4.3 Dynamic Groups

In the same way you can add windows to an application after its creation, you can add elements to already existing group objects. This may be useful if a group contains many similar children or if the number of children is not known in the beginning.

You can add new elements to groups or delete them again, but the window that contains this group must not be open!

A small example:

```
    app = ApplicationObject,
        ...,
        SubWindow, win = WindowObject,
            WindowContents, VGroup,
                ...,
                grp = VGroup,
            End,
            ...,
        End,
    End,
```

```
End;

/* The group 'grp' has been created without any children. */
/* The window must not be opened now!                     */

for (i=0; i<NumPlayers; i++)
{
   Object *name = StringObject, MUIA_String_MaxLen, 30, End;
   if (name)
      DoMethod(grp,OM_ADDMEMBER,name); // add gadget to group.
   else
      fail();
}

/* After we have at least one element in the group~'grp', */
/* the window can be opened...                            */
```

Of course you may (if the window is closed) remove elements from groups. Please note that window objects containing empty groups must not be opened.

5 Custom Classes

5.1 Introduction

MUI features a lot of builtin classes that already allow creation of powerful applications. However, a generic GUI system will never be able to satisfy all the requirements of all kinds of programs. A sound editor would e.g. need a class to display and edit sound data, a paint program would need a drawing area and a chess game would need a chess-board class.

Prior to MUI V2.0, programmers were stuck with the builtin classes. Except of the (sometimes problematic) BOOPSI interface, there was no way to include custom gadgets in a MUI window. This big disadvantage has finally disappeared, with MUI 2.0 it's posssible to write and use private classes just like one of the builtins.

Beneath the BOOPSI gadget interface, private classes are the only way to have custom gadgets in a MUI window. Drawing into windows directly from the applications task is illegal and will surely lead into lots of problems!

Note: Since this manual doesn't repeat the RKMs BOOPSI section, you should be familiar with the BOOPSI mechanisms of dispatchers, methods, attributes, instance data structures, etc. before going on.

5.2 Overview

Building a class is rather simple. All you have to do is to write a class dispatcher function, setup a hook structure, call MUI_GetClass() to find the pointer to your superclass and use intuition.library/MakeClass() to create the class. If you already wrote BOOPSI classes, you should know whats going on here.

Objects from custom classes are created using intuition.library / NewObject(), not muimaster.library / MUI_NewObject(). Nevertheless, you can use these objects like every other MUI objects, e.g. as children of groups, in virtual groups or wherever you like.

All classes you create have to be subclasses of MUI's area class, most of them will probably be direct subclasses but you can also have subclasses of list class or even subclasses of your own classes if you want. Here's an example of how a custom class could be generated:

```
/* Get a pointer to the superclass. MUI will lock this */
/* and prevent it from being flushed during you hold   */
/* the pointer. When you're done, you have to call     */
/* MUI_FreeClass() to release this lock.               */

if (!(SuperClass=MUI_GetClass(MUIC_Area)))
   fail("Superclass for the new class not found.");

/* Create the new class with the boopsi function call. */
/* You will need the sizeof your classes instance data */
/* here.                                               */

if (!(MyClass = MakeClass(NULL,NULL,SuperClass,sizeof(struct MyData),0)))
{
   MUI_FreeClass(SuperClass);
   fail("Failed to create class.");
}

/* Set the dispatcher for the new class. */

MyClass->cl_Dispatcher.h_Entry    = (APTR)MyDispatcher;
MyClass->cl_Dispatcher.h_SubEntry = NULL;
MyClass->cl_Dispatcher.h_Data     = NULL;

...
...

app = ApplicationObject,
   ...,

   SubWindow, window = WindowObject,
      ...
      WindowContents, VGroup,

         Child, MyObj = NewObject(MyClass,NULL,
            TextFrame,
            MUIA_Background, MUII_BACKGROUND,
            TAG_DONE),

         End,

      End,
   End;

...
...

/* Shutdown. When the application is disposed,
/* MUI also deletes MyObj. */
```

```
   MUI_DisposeObject(app);     // dispose all objects.
   FreeClass(MyClass);         // free our private class.
   MUI_FreeClass(SuperClass);  // release super class pointer.
```

You're dispatcher will look like a traditional BOOPSI dispatcher:

```
SAVEDS ASM ULONG MyDispatcher(
   REG(a0) struct IClass *cl,
   REG(a2) Object *obj,
   REG(a1) Msg msg)
{
   switch (msg->MethodID)
   {
      case OM_NEW        : return(mNew      (cl,obj,(APTR)msg));
      case OM_DISPOSE    : return(mDispose  (cl,obj,(APTR)msg));
      case MUIM_AskMinMax: return(mAskMinMax(cl,obj,(APTR)msg));
      case MUIM_Draw     : return(mDraw     (cl,obj,(APTR)msg));
      ...
   }

   return(DoSuperMethodA(cl,obj,msg));
}
```

What methods are available and need to be supported is discussed in the following chapter.

Note: Your dispatcher will also receive some undocumented methods. It's not a wise choice to make any assumptions here, the only thing you should do is pass them to your superclass immediately!

5.3 Methods

The MUI system talks to its classes with a specific set of methods. Some of these methods need to be implemented by all MUI classes but most of them are optional. This chapter gives a quick overview about the methods your class might receive. All methods are discussed more detailed in the following sections.

As usual with BOOPSI objects, you will get an OM_NEW whenever a new object of your class shall be created. Since with MUI, the deepest nested objects are always created first, your object won't know anything about its display environment during OM_NEW.

After all objects are created, MUI finds out about the display environment (screen, drawinfo structure, fonts, etc.) and sends you a MUIM_Setup method with this information. You can calculate some internal data here that depends on the display enviroment (e.g. the line height in an editor class). Note that you still don't have a intuition window at this point.

The next method your class receives is called MUIM_AskMinMax. MUI wants to find out about your minimum, maximum and default sizes to prepare its window size calculation and layout algorithms. Since you already know about display environment, calculating these values should be easy at this point.

After asking all objects about their dimensions and adding the results depending on the type of group your object resides in, MUI is able to open the window. Once it is open, your object will receive a MUIM_Show method telling you that you are about to be added to the window. MUIM_Show is mainly intended for use by intuition like gadgets, they will do an AddGadget() here (and a RemGadget() during MUIM_Hide). Usually, you won't need to implement these methods.

Since your class still didn't draw anything, its time for a MUIM_Draw method now. MUI sends this method whenever it feels that you should draw yourself. This happens of course after a window has been opened but also when you reside in a simple refresh window and need to be refreshed. MUIM_Draw is the only place where you are actually allowed to draw something!

When the window is resized, MUI sends a MUIM_Hide (allowing some RemGadget() for intuition like gadgets), calculates new positions and sizes and sends a MUIM_Show and a MUIM_Draw again. However, if you correctly implement MUIM_Draw, you don't need to care about this in almost all cases.

When your window is about to be closed (either because your application no longer needs it or because the user iconified your program or changed the preferences settings), you will receive a MUIM_Cleanup. This allows you to free some display environment dependant things you allocated during MUIM_Setup.

The last thing your object will get is of course a OM_DISPOSE after it has hopefully done all the things you wanted it to do.

To sum it up again, look at the diagram below. Things between brackets might be called zero or more times for the same object.

```
OM_NEW; /* you dont know anything about display environment here */
```

```
{
   MUIM_Setup;      /* information about display, still no window */
   MUIM_AskMinMax;  /* tell me your min/max dimensions */
   [ window is opened here ]
   {
      MUIM_Show;    /* add yourself to the window, don't yet draw */
      {
         MUIM_Draw;     /* draw yourself */
      }
      MUIM_Hide;    /* remove yourself from the window */
   }
   [ window is closed here ]
   MUIM_Cleanup;    /* free any display dependant data */
}
OM_DISPOSE; /* kill yourself completely */
```

As you probably noticed, most methods are implemented as constructor/destructor pairs: OM_NEW & OM_DISPOSE, MUIM_Setup & MUIM_Cleanup, MUIM_Show & MUIM_Hide. For every constuctor call, you will receive exactly one destructor call. Usually, you will allocate some resources during OM_NEW, MUIM_Setup or MUIM_Show and free these resources during OM_DISPOSE, MUIM_Cleanup or MUIM_Hide respectively.

The three levels of constructor/destructor pairs feature different amount of available information about the display environment.

– OM_NEW (destructor OM_DISPOSE)

  No information at all. All you can do is parse the initial attribute list and store its contents in the instance data structure of your object.

– MUIM_Setup (destructor MUIM_Cleanup)

  MUI has figured out what screen and font to use. You can allocate/calculate things that depend on this information.

– MUIM_Show (destructor MUIM_Hide)

  MUI has finally opened an intuition window at this point. You are able to allocate resources depending on a window context here.

You can rely on the fact that you receive OM_NEW before MUIM_Setup (of course, otherwise you wouldn't have a valid object pointer) and MUIM_Setup before MUIM_Show. But always keep in mind that you might get multiple MUIM_Show / MUIM_Hide pairs or MUIM_Setup / MUIM_Cleanup pairs or that you could receive a MUIM_Setup / MUIM_Cleanup pair without some MUIM_Show / MUIM_Hide between.

5.4 OM_NEW & OM_DISPOSE

Implementing these methods for MUI objects is mainly identical to traditional BOOPSI objects. The only thing important to remember is that MUI objects have really no idea about display environment or other configuration issues when receiving OM_NEW. The ops_GInfo member of the opSet parameter structure is always unused. Typical new/dispose methods consist of some tag list parsing and instance data initialisation.

```
static ULONG mNew(struct IClass *cl,Object *obj,struct opSet *msg)
{
   struct INST_DATA *data;

   if (!(obj = (Object *)DoSuperMethodA(cl,obj,msg)))
      return(0);

   data = INST_DATA(cl,obj);

   data->min = GetTagData(MYATTR_Min,  0,msg->ops_AttrList);
   data->max = GetTagData(MYATTR_Max,100,msg->ops_AttrList);
   data->lvl = GetTagData(MYATTR_Lvl, 50,msg->ops_AttrList);

   data->count  = data->max - data->min + 1;
   data->buffer = NULL;

   if (data->count > 0)
   {
      if (data->buffer = AllocVec(data->count * 4))
      {
         return((ULONG)obj);
      }
   }

   /* invoke OM_DISPOSE on *our* class! */
   CoerceMethod(cl,obj,OM_DISPOSE);
   return(0);
}


static ULONG mDispose(struct IClass *cl,Object *obj,Msg msg)
{
   struct Data *data = INST_DATA(cl,obj);

   if (data->buffer) FreeVec(data->buffer);

   return(DoSuperMethodA(cl,obj));
}
```

5.5 MUIM_Setup & MUIM_Cleanup

Since your object doesn't know anything about display environment after it is created with OM_NEW, MUI will send you an MUIM_Setup when it is about to open a window containing your object.

The first thing you have to do is to pass MUIM_Setup to your super class an return FALSE on failure. After this, you can calculate some internal data or allocate some display buffers. Return TRUE if everything went ok or FALSE when you discovered any errors. In this case, MUI will send a MUIM_Cleanup to all objects that already received a MUIM_Setup (excluding the current one!) and fail to open the window. Typical setup/cleanup pairs could look like this:

```
static ULONG mSetup(struct IClass *cl,Object *obj,Msg msg)
{
   struct Data *data = INST_DATA(cl,obj);

   if (!(DoSuperMethodA(cl,obj,msg)))
      return(FALSE);

   data->lineheight = _font(obj)->tf_YSize;

   if (!(data->linebuf = AllocVec(data->count*data->lineheight)))
      return(FALSE);

   MUI_RequestIDCMP(obj,IDCMP_RAWKEY|IDCMP_MOUSEBUTTONS|IDCMP_INACTIVEWINDOW);

   return(TRUE);
}


static ULONG mCleanup(struct IClass *cl,Object *obj,Msg msg)
{
   struct Data *data = INST_DATA(cl,obj);

   FreeVec(data->linebuf);

   MUI_RejectIDCMP(obj,IDCMP_RAWKEY|IDCMP_MOUSEBUTTONS|IDCMP_INACTIVEWINDOW);

   return(DoSuperMethodA(cl,obj));
}
```

5.6 MUIM_AskMinMax

With MUIM_AskMinMax, MUI wants to find out about the minimum, maximum and default sizes of your object. These values are needed for the correct layout, depending on the type of group that contains your object.

```
static ULONG mAskMinMax(
   struct IClass *cl,
   Object *obj,
   struct MUIP_AskMinMax *msg)
{
   /*
   ** let our superclass first fill in what it thinks about sizes.
   ** this will e.g. add the size of frame and inner spacing.
   */

   DoSuperMethodA(cl,obj,msg);

   /*
   ** now add the values specific to our object. note that we
   ** indeed need to *add* these values, not just set them!
   */

   /* x-size depending on objects font */
   msg->MinMaxInfo->MinWidth  += _font(obj)->tf_XSize * 10;
   msg->MinMaxInfo->DefWidth  += _font(obj)->tf_XSize * 20;
   msg->MinMaxInfo->MaxWidth  += MAXMAX; /* unlimited */

   /* fixed y-size */
   msg->MinMaxInfo->MinHeight += _font(obj)->tf_YSize;
   msg->MinMaxInfo->DefHeight += _font(obj)->tf_YSize;
   msg->MinMaxInfo->MaxHeight += _font(obj)->tf_YSize;

   return(0);
}
```

5.7 MUIM_Show & MUIM_Hide

Once the window is opened, your object will receive a MUIM_Show. If you have some window/rastport environment dependant things to do, MUIM_Show is the correct place. Intuition like gadgets would for example do an AddGadget() here.

Note that you should **not** render during MUIM_Show. Usually, MUI classes won't need to implement this method.

5.8 MUIM_Draw

Whenever MUI feels that your object should render itself, it sends you a MUIM_Draw method. This happens e.g. when a window is openend for the first time, after a window was resized or when a simple refresh window needs to be refreshed. In the latter case, MUI already set up a clip region to restrict rendering to the necessary areas.

Together with MUIM_Draw comes a flag value that indicates which parts of the object are to be redrawn. The only interesting bits in this flag value are MADF_DRAWOBJECT and MADF_UPDATE. When MADF_DRAWOBJECT is set, MUI wants you to do a complete redraw of your object. MADF_UPDATE is not used by the MUI system itself but is reserved for your private requirements width the MUI_Redraw() function call. See the example programs coming with the MUI distribution for details.

Information about rendering environment (screen, window, rastport, pens, etc.) is saved in a structure called MUI_RenderInfo. Every objects render info structure is accessable with the muiRenderInfo(obj) macro. Parts of this structure are valid between MUIM_Setup and MUIM_Cleanup, other parts like window and rastport pointer are valid between MUIM_Show and MUIM_Hide. Please have a look at the supplied compiler headers for more detailed information about the MUI_RenderInfo structure.

Note: MUIM_Draw is the only place where you are allowed to render!

5.9 MUIM_HandleInput

Compared with BOOPSI, MUI uses a different input handling scheme. Not only the "active" object but instead all objects may receive input events at the same time. Since sending input events to every object in a window would be an incredible overhead, you have to specify what messages you really need. MUI offers the library calls MUI_RequestIDCMP() and MUI_RejectIDCMP() for this purpose. Whenever an arriving input event matches your request, your object will receive a MUIM_HandleInput method.

You can call MUI_RequestIDCMP() and MUI_RejectIDCMP() at every time. Performance affecting critical events like INTUITICKS and MOUSEMOVEs should only be requested when you

really need them. The "Class3" for example requests MOUSEBUTTONS and RAWKEY during MUIM_Setup. The critical MOUSEMOVES are only requested when a button was pressed and immediately rejected after it was released again.

Beneath the struct IntuiMessage, MUIM_HandleInput receives a longword describing a MUIKEY as second parameter. If this is set to some other value as MUIKEY_NONE, your object is the actve object and the input event translated to a user configured keyboard action.

MUI will **not** translate input events to your objects coordinates. This is up to you. A typical input implementation could look like this:

```
static ULONG mHandleInput(
   struct IClass *cl,
   Object *obj,
   struct MUIP_HandleInput *msg)
{
   #define _between(a,x,b) ((x)>=(a) && (x)<=(b))
   #define _isinobject(x,y) (_between(_mleft(obj),(x),_mright (obj))
                          && _between(_mtop(obj) ,(y),_mbottom(obj)))

   struct Data *data = INST_DATA(cl,obj);

   if (msg->muikey)
   {
      switch (msg->muikey)
      {
         case MUIKEY_LEFT :
            data->sx=-1;
            MUI_Redraw(obj,MADF_DRAWUPDATE);
            break;
         case MUIKEY_RIGHT:
            data->sx= 1;
            MUI_Redraw(obj,MADF_DRAWUPDATE);
            break;
         case MUIKEY_UP   :
            data->sy=-1;
            MUI_Redraw(obj,MADF_DRAWUPDATE);
            break;
         case MUIKEY_DOWN :
            data->sy= 1;
            MUI_Redraw(obj,MADF_DRAWUPDATE);
            break;
      }
   }
```

```
    if (msg->imsg)
    {
       switch (msg->imsg->Class)
       {
          case IDCMP_MOUSEBUTTONS:
          {
             if (msg->imsg->Code==SELECTDOWN)
             {
                if (_isinobject(msg->imsg->MouseX,msg->imsg->MouseY))
                {
                   data->x = msg->imsg->MouseX;
                   data->y = msg->imsg->MouseY;
                   MUI_Redraw(obj,MADF_DRAWUPDATE);
                   MUI_RequestIDCMP(obj,IDCMP_MOUSEMOVE);
                }
             }
             else
                MUI_RejectIDCMP(obj,IDCMP_MOUSEMOVE);
          }
          break;

          case IDCMP_MOUSEMOVE:
          {
             if (_isinobject(msg->imsg->MouseX,msg->imsg->MouseY))
             {
                data->x = msg->imsg->MouseX;
                data->y = msg->imsg->MouseY;
                MUI_Redraw(obj,MADF_DRAWUPDATE);
             }
          }
          break;
       }
    }

    /* passing MUIM_HandleInput to the super class is only necessary
       if you rely on area class input handling (MUIA_InputMode). */

    return(0);
}
```

5.10 OM_SETGET

Implementing OM_SET and OM_GET is similar to oldstyle BOOPSI gadgets. The only important thing to know about is that you should **not** render in a OM_SET (e.g. as a result of an attribute

change). Instead, call MUI_Redraw() with a MADF_DRAWOBJECT or a MADF_UPDATE flag, MUI will then call your objects Draw method.

```
static ULONG mSet(struct IClass *cl,Object *obj,Msg msg)
{
   struct MyData *data = INST_DATA(cl,obj);
   struct TagItem *tags,*tag;

   for (tags=((struct opSet *)msg)->ops_AttrList;tag=NextTagItem(&tags);)
   {
      switch (tag->ti_Tag)
      {
         case MYATTR_PEN:
            data->pen = tag->ti_Data;         /* set the new value */
            MUI_Redraw(obj,MADF_DRAWOBJECT);  /* complete redraw */
            break;

         case MYATTR_LEVEL:
            data->level = tag->ti_Data;       /* set the new value */
            MUI_Redraw(obj,MADF_DRAWUPDATE);  /* only update ourselves */
            break;
      }
   }

   return(DoSuperMethodA(cl,obj,msg));
}
```

5.11 CSC_DISTRIBUTE

Usually, you will use custom classes only for your own applications. In this case, you won't need to care about the tag values used for your private attributes and methods. The only thing you should consider is that all standard MUI classes use values betwenn 0x80420000 and 0x8042ffff for their tags. To avoid conflicts, all you have to do is make your tags start with anything but 0x8042.

However, if you start distributing your classes to make other people benefit from your work and help them in writing better MUI applications, things get a bit more complicated. MUI will get confused if two or more classes start using some equal tag values. To avoid these problems, I suggest to use your MUI serial number together with the TAG_USER bit as upper word for your tag items. Thus, if your serial number is e.g. 123, all your tag items would look like

```
#define MUIA_Myclass_Foobar     (TAG_USER | (123<<16) | 0x0000)
#define MUIA_Myclass_Barfoo     (TAG_USER | (123<<16) | 0x0001)
```

```
#define MUIA_Myclass_Deadbeaf   (TAG_USER | (123<<16) | 0x0002)
#define MUIM_Myclass_Doit       (TAG_USER | (123<<16) | 0x0003)
#define MUIM_Myclass_Doit2      (TAG_USER | (123<<16) | 0x0004)

#define MUIA_Myotherclass_Attr1 (TAG_USER | (123<<16) | 0x0010)
#define MUIA_Myotherclass_Attr2 (TAG_USER | (123<<16) | 0x0011)
#define MUIA_Myotherclass_Attr3 (TAG_USER | (123<<16) | 0x0012)
#define MUIA_Myotherclass_Attr4 (TAG_USER | (123<<16) | 0x0013)
```

If you aren't registered and don't yet have a serial number, no problem... just register **now**! ;-)

5.12 CSC_LIBRARIES

From version 2.0 on, MUI supports the creation of external custom class libraries. See the autodocs of MUI_CreateCustomClass() and MUI_DeleteCustomClass() and the supplied demo classes for details.

6 Style Guide

6.1 Overview

Note: These topics aren't discussed here just for fun. You will annoy lots of users if you don't pay attention to them!

- File Requester

  Even if MUI features a file list and a volume list object and makes building a private file requester very easy, you should always provide a possibility to pop up a standard asl requester for this purpose. Just add a little popup button right beneath your file string gadget and everything will be fine. MUI offers a file-popup object exactly for this purpose. Note well: Many users (including myself) move programs with non-standard file requesters into the trashcan immediately.

- Window Size

  With MUI, it's very easy to have lots of gadgets within a single window. Since you as a programmer usually have a more powerful system with higher graphic resolutions as most of your users, windows tend to become too big. You should always make sure that everything you design fits on a standard 640x256 screen with a topaz/8 font. Otherwise, MUI will try to use very small fonts or virtual groups to make your window fit, making your application look and feel bad.

- Keyboard Control

  Even if you're a "mouse-only" user, add keyboard cycle chains and gadget shortcuts to your application. It's very few work for you and helps lots of users.

- Background

  MUI allows the user to adjust lots of different backgrounds for objects. Even if you don't use this feature, you should always test your program with a fancy background pattern configuration and check whether all your buttons really have button backgrounds, all your framed texts really have text backgrounds, etc.

- and last...

  Don't forget the traditional Amiga style guide!