MUI/Developer/Docs/MUIdev

COLLABORATORS

	TITLE : MUI/Developer/Docs/MUIdev		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		December 4, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

MUI/Developer/Docs/MUIdev

1.1 MUI/Developer/Docs/MUIdev.guide

```
                    MUI - MagicUserInterface

     A system to create and maintain graphical user interfaces

                    Programmer Documentation

               (c) Copyright 1993 by Stefan Stuntz


Getting Started...

  OOP Overview                    Some words about Boopsi.
  Available Classes               List of available classes.
  Application Theory              The application tree.
  Object Handling                 General object handling.
  Macros                          Creating objects with macros.

Layout...

  Basics                          Automatic layout engine.
  Groups                          Grouping objects.

Building An Appliccation...

  Creation                        Object generation.
  Notification                    Object connections.

Dynamic Object Linking...

  Overview                        OM_ADDMEMBER and OM_REMMEMBER.
  Windows                         Dynamic creation of windows.
  Groups                          Dynamic creation of gadgets.
```

1.2 MUIdev.guide/GST_OOP

```
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!

```
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.

1.3 MUIdev.guide/GST_CLASSES

```
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)
```

1.4 MUIdev.guide/GST_APPTHEORY

```
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
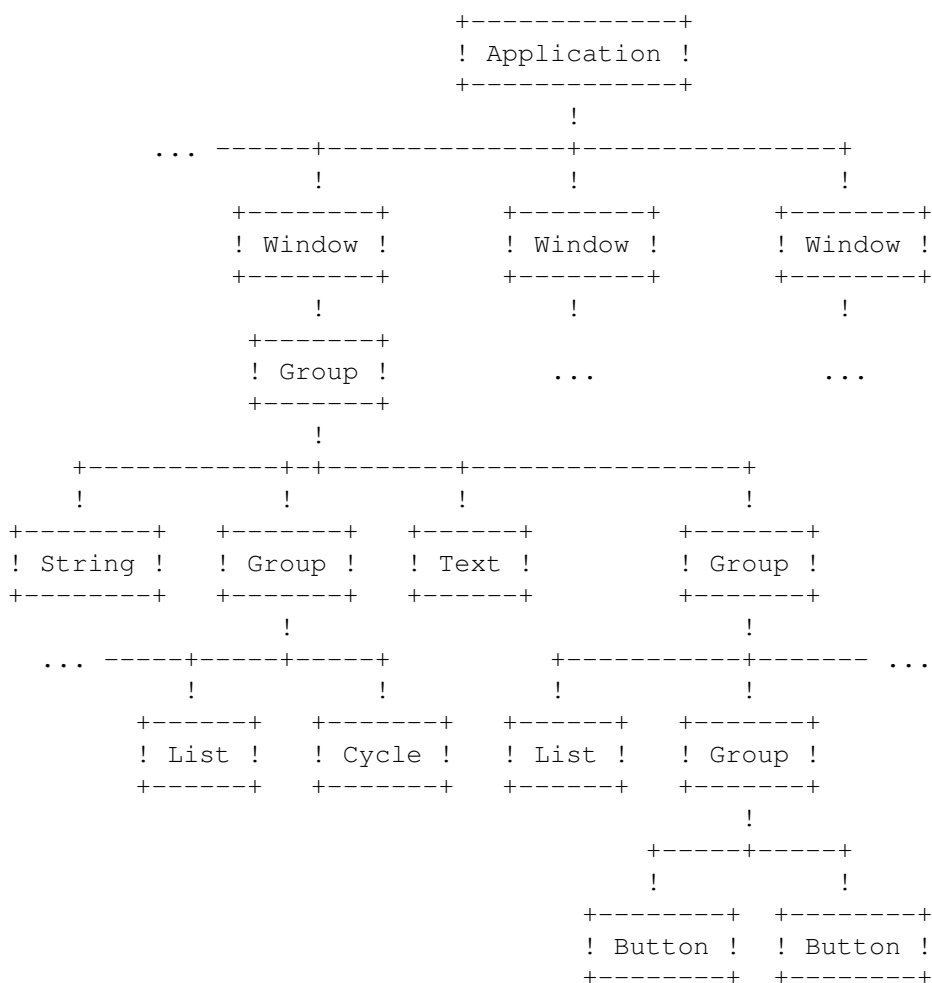always need a so called root object.

   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:

```
                              +-------------+
                              ! Application !
                              +-------------+
                                     !
          ... -----+--------------+---------------+
               !              !               !
          +--------+      +--------+      +--------+
          ! Window !      ! Window !      ! Window !
          +--------+      +--------+      +--------+
               !              !               !
          +-------+          ...             ...
          ! Group !
          +-------+
               !
     +------------+-+--------+----------------+
     !            !          !                !
 +--------+   +-------+   +------+        +-------+
 ! String !   ! Group !   ! Text !        ! Group !
 +--------+   +-------+   +------+        +-------+
                  !                           !
     ... -----+-----+-----+         +----------+------- ...
          !         !     !         !          !
      +------+  +-------+ +------+  +-------+
      ! List !  ! Cycle ! ! List !  ! Group !
      +------+  +-------+ +------+  +-------+
                                       !
                                  +-----+-----+
                                  !           !
                              +--------+  +--------+
                              ! Button !  ! Button !
                              +--------+  +--------+
```

   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.
```

1.5 MUIdev.guide/GST_OBJECTS

```
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.

1.6 MUIdev.guide/GST_MACROS

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.

1.7 MUIdev.guide/LAY_BASICS

Layout Engine
****

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.

1.8 MUIdev.guide/LAY_GROUPS

Groups
======

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

```
+---------------------------------------+
!                                       !
!  +----------------------+ +------+  !
!  ! C              (dir) ! ! dh0: !  !
!  ! Classes         (dir) ! ! dh1: !  !
!  ! Devs            (dir) ! ! dh2: !  !
!  ! Expansion       (dir) ! ! df0: !  !
!  ! ...                  ! ! df1: !  !
!  ! Trashcan.info   1.172 ! ! df2: !  !
!  ! Utilities.info    632 ! ! ram: !  !
!  ! WBStartup.info    632 ! ! rad: !  !
!  +----------------------+ +------+  !
!                                       !
!  Path: _____   !
!                                       !
!  File: _____   !
!                                       !
!  +------+                  +--------+  !
!  ! Okay !                  ! Cancel !  !
!  +------+                  +--------+  !
!                                       !
+---------------------------------------+
```

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 a 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:

```
       +-------------------------------+
       !                               !
       !   +-------+ +-------+ +-------+   !
       !   !       ! !       ! !       !   !
       !   ! field ! ! field ! ! field !   !
       !   !   1   ! !   2   ! !   3   !   !
       !   !       ! !       ! !       !   !
       !   +-------+ +-------+ +-------+   !
       !                               !
       +-------------------------------+
```

   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.

1.9 MUIdev.guide/APP_CREATE

```
Building An Application
***********************

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.

1.10 MUIdev.guide/APP_NOTIFY

```
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 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.

1.11 MUIdev.guide/DYN_OVERVIEW

```
Dynamic Object Linking
**********************
```

```
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!

1.12 MUIdev.guide/DYN_WINDOWS

```
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
```

1.13 MUIdev.guide/DYN_GROUPS

```
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.