

AmigaMail

COLLABORATORS

	<i>TITLE :</i> AmigaMail		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY		July 19, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1	AmigaMail	1
1.1	IV-41: Writing a Boopsi Image Class	1
1.2	Callback Hooks	2
1.3	Initializing a Boopsi Class	3
1.4	Boopsi Dispatcher	4
1.5	Boopsi Rootclass Methods	5
1.6	OM_NEW	6
1.7	OM_DISPOSE	7
1.8	OM_SET	7
1.9	OM_GET	8
1.10	Imageclass Methods	8
1.11	IM_DRAW	8
1.12	IM_ERASE	9
1.13	IM_HITTEST	10
1.14	IM_DRAWFRAME	10
1.15	IM_ERASEFRAME	11
1.16	IM_HITFRAME	11
1.17	IM_MOVE	12
1.18	IM_FRAMEBOX	12
1.19	Image Class Example	13

Chapter 1

AmigaMail

1.1 IV-41: Writing a Boopsi Image Class

By David N. Junod

Editor's note: this article and its source code reference several functions that, at press time, were only available from the classface.asm and hookface.asm assembly source files that appear on the Atlanta and Milan DevCon disks. The functions from these files should eventually appear in amiga.lib.

The most sophisticated level of Intuition programming is to write a boopsi class. The easiest way to enter the boopsi class writer's arena is to write an image class.

Boopsi's imageclass is one of the standard classes built into Intuition. As its name implies, it is a class of Intuition Images. These boopsi images can be used in place of traditional Image structure (as they contain an Intuition Image structure), but they are much more powerful. By using boopsi methods, an application or Intuition can tell an imageclass object to render itself. Because it renders itself (rather than Intuition rendering it), the imageclass object is free to render whatever it wants (well, within reason). For example, a boopsi image object can render itself according to the current display resolution, or to scale itself to any size an application requests.

This article is designed to provide the novice boopsi programmer with the information needed to write an image class for their application. This article assumes the reader is already familiar with some boopsi concepts. For more information on boopsi, see the article "Introduction to Boopsi" in the March/April 1991 issue of Amiga Mail or the Atlanta (or Milan) DevCon notes and disks. The example custom class at the end of this article, mytextlabelclass.c, shows how to create a custom image class that renders a text label with an underline beneath a character. This character can be used to trigger some event.

When designing a specific class, you must first choose a superclass that is suitable for your needs. If you are creating a new image class, then its superclass will either be imageclass or some subclass of imageclass.

Classes may be public or private. Any application can access a public class. Before a class can be considered public it must first have a name and must be part of the public class list maintained by Intuition. It can then be accessed by other applications. A private class is not in the public class list and can only be used by applications that have a pointer to the Class structure (usually the application that implemented the class).

Callback Hooks	Boopsi Rootclass Methods
Initializing a Boopsi Class	Imageclass Methods
Boopsi Dispatcher	Image Class Example

1.2 Callback Hooks

When you present a custom image to Intuition, you provide a pointer to a Hook structure that Intuition uses to find functions needed by various image operations. Without going into great detail, a hook provides a pointer to a function that the system calls using Amiga register parameter conventions. The hook supplies enough information to conveniently transfer control to a High-Level Language (HLL) entry point. Boopsi imageclass objects provide Intuition with a hook to a method dispatcher function.

The Hook structure is defined as follows (from <utility/hooks.h>):

```
/* new standard hook structure */
struct Hook
{
    struct MinNode  h_MinNode;
    ULONG          (*h_Entry) ();    /* stub function entry point */
    ULONG          (*h_SubEntry) (); /* the custom function entry point */
    VOID           *h_Data;          /* owner specific */
};
```

The assembly language stub for C parameter conventions that boopsi classes (and custom gadgets) require is:

```
_hookEntry:
    move.l  a1,-(sp)           ; push message packet pointer
    move.l  a2,-(sp)           ; push object pointer
    move.l  a0,-(sp)           ; push hook pointer
    move.l  h_SubEntry(a0),a0   ; fetch C entry point ...
    jsr     (a0)               ; ... and call it
    lea     12(sp),sp          ; fix stack
    rts
```

The C language stub, for C compilers that support registerized parameters is:

```
/* This function converts register-parameter hook calling
 * convention into standard C conventions. It requires a C
 * compiler that supports registerized parameters, such as
 * SAS/C 5.xx or greater.
 */
ULONG __asm hookEntry(
```

```

    register __a0 struct Hook *h,
    register __a2 VOID *o,
    register __a1 VOID *msg)
{
    return ((*h->h_SubEntry)(h, o, msg));
}

```

1.3 Initializing a Boopsi Class

You need some simple code to initialize a class and its hook. When initializing a class, you specify the size of the class's instance and what the superclass is, and you also have to supply a pointer to a hook entry stub.

The following code fragment illustrates the initialization of a private subclass of imageclass.

```

ULONG __saveds dispatchmyTextLabel();

/* This is the data that each instance of our class will need. */
struct localObjData
{
    /* Font to use */
    struct TextFont *lod_Font;

    /* The key that is underlined */
    UWORD lod_Key;

    /* DrawMode */
    UBYTE lod_Mode;
};

#define MYCLASSID          NULL
#define SUPERCLASSID      (IMAGECLASS)
#define LSIZE              (sizeof(struct localObjData))

Class *initmyTextLabelClass (VOID)
{
    extern ULONG __saveds dispatchmyTextLabel();
    extern ULONG hookEntry ();          /* defined in hookface.asm */
    Class *cl;

    if (cl = MakeClass (MYCLASSID, SUPERCLASSID, NULL, LSIZE, 0))
    {
        /* Fill in the callback hook */
        cl->cl_Dispatcher.h_Entry = hookEntry;
        cl->cl_Dispatcher.h_SubEntry = dispatchmyTextLabel;
    }

    /* Return a pointer to the class */
    return (cl);
}

```

In order to make the class public instead of private, do the following:

```
#define MYCLASSID          "mytextlabelclass"

ULONG __saves dispatchmyTextLabel();

Class *initmyTextLabelClass (VOID)
{
    extern ULONG __saves dispatchmyTextLabel();
    extern ULONG hookEntry ();
    Class *cl;

    if (cl = MakeClass (MYCLASSID, SUPERCLASSID, NULL, LSIZE, 0))
    {
        /* Fill in the callback hook */
        cl->cl_Dispatcher.h_Entry = hookEntry;
        cl->cl_Dispatcher.h_SubEntry = dispatchmyTextLabel;

        /* Make the class public */
        AddClass (cl);
    }

    /* Return a pointer to the class */
    return (cl);
}
```

1.4 Boopsi Dispatcher

Now all you need to do is implement a dispatcher routine. When the dispatcher is in operation, Intuition passes method IDs to it. The dispatcher will either execute code corresponding to the a method ID (the code is usually part of the dispatcher) or delegate processing the method to the superclass (or it can do a little of both).

The following fragment provides an example of what a dispatcher for a boopsi class looks like (Note that __saves (Save DS) is used to insure that register A4 is set up properly for indirect addressing with the SASC compiler):

```
ULONG __saves dispatchmyTextLabel (Class *cl, Object *o, Msg msg)
{
    struct localObjData *lod;
    Object *newobj;
    ULONG retval;

    switch (msg->MethodID)
    {
        /* Create a new object */
        case OM_NEW:
            /* Have our superclass create it. DSM() passes on the
             * message to the superclass, where msg is the structure
             * containing the message specific parameters.
             */
            /* ... */
    }
}
```

```

        if (newobj = (Object *) DSM (cl, o, msg))
        {
            /* Set the attributes */
            setmyTextLabelAttrs(cl, newobj, (struct opSet *) msg);
        }

        retval = (ULONG) newobj;
        break;

/* Obtain information on an attribute */
case OM_GET:
    retval = getmyTextLabelAttrs (cl, o, (struct opGet *) msg);
    break;

/* Set attributes */
case OM_UPDATE:
case OM_SET:
    /* Let the superclass set the attributes that it
     * knows about. */
    retval = DSM (cl, o, msg);

    /* Set the attributes that we care about */
    retval |= setmyTextLabelAttrs (cl, o, (struct opSet *) msg);
    break;

/* Draw the various states that the image supports */
case IM_DRAW:
case IM_DRAWFRAME:
    retval = drawmyTextLabel (cl, o, (struct impDraw *) msg);
    break;

/* Let the superclass handle everything else */
default:
    retval = (ULONG) DSM (cl, o, msg);
    break;
}

return (retval);
}

```

1.5 Boopsi Rootclass Methods

Since all classes should be subclasses of some class, with the exception of rootclass, all classes you write will be subclasses--perhaps indirectly so--of rootclass. Because of this, your class must either implement the rootclass methods or defer processing of these methods to the superclass (as DispatchmyTextLabel() did). Provided below are brief descriptions of the rootclass methods. Remember that any message unrecognized by a class dispatcher should be passed to the superclass (using the amiga.lib functions DSM() or DoSuperMethod()).

The rootclass method IDs that a subclass of imageclass needs to understand are:

OM_NEW	Create a new object.
--------	----------------------


```

OM_DISPOSE    Delete an object.
OM_SET        Change an object's attributes.
OM_GET        Retrieve the value of one of the object's attributes.

```

The dispatcher should pass other rootclass methods on to the superclass.

Each method requires one or more parameters. The MethodID is the only common parameter for each method.

```

OM_NEW        OM_DISPOSE        OM_SET        OM_GET

```

1.6 OM_NEW

The OM_NEW method receives the following arguments:

```

struct opSet
{
    ULONG MethodID;
    struct TagItem *ops_AttrList;
    struct GadgetInfo *ops_GInfo;
};

```

The ops_AttrList field contains a pointer to the TagItem array of attributes for the new object. The ops_GInfo field is always NULL for the OM_NEW method.

Unlike other methods, this method is not passed an object pointer (since the whole idea is to create an object). The pointer normally used to pass a boopsi object is instead used to pass the address of the object's "true class" (the class the object is an instance of). That way, all class dispatchers can tell if they are the "true class" of the object being created (as opposed to a superclass of the true class). Also, with this pointer, rootclass can determine what the instance data is for an object, and can allocate the right amount of memory for it.

For the OM_NEW method, the new class's dispatcher should do the following:

- 1) Pass the message along to the superclass. All classes do this as rootclass takes care of allocating memory for the new object. As the OM_NEW method works "top down" (from rootclass down through its subclasses to the true class), each class will in turn initialize its corresponding instance data. This all happens before the new class's dispatcher regains control. Eventually, the message comes back from the superclass with a newly allocated object (unless of course something failed and you receive a NULL pointer instead).
- 2) Obtain a pointer to the object's instance data for this class. Use the INST_DATA() macro (defined in <intuition/classes.h>). INST_DATA() takes two arguments, a pointer to your class and a pointer to the object.

```

void *INST_DATA(*Class, *Object);

```

- 3) Initialize your instance data. You may allocate additional memory buffers for your object, or even create other objects which are

components to objects of your class.

- 4) Process your initial attribute list (from the opSet structure passed in the OM_NEW message). In particular, process all the attributes that can be set only at initialization time. After you deal with the "initialization only" attributes, apply the same attribute processing on these remaining attributes that you would apply to an OM_SET message.
- 5) Return the object to the caller.

1.7 OM_DISPOSE

The OM_DISPOSE method instructs the class to deallocate an object. This method receives no parameters.

For the OM_DISPOSE method, the new class's dispatcher should do the following:

- 1) Free any additional memory you allocated (memory allocated in step 3 from OM_NEW).
- 2) Dispose of any objects that you created as components of your object (component objects created in step 3 from OM_NEW).
- 3) Pass the message up to the superclass, which will eventually reach rootclass, which will free the memory allocated for the object.

The mytextlabelclass example at the end of this article does not allocate any extra resources when it creates an object. Because it does not have to release any resources, the mytextlabelclass dispatcher lets its superclass handle the OM_DISPOSE method. Eventually, some superclass of mytextlabelclass will deallocate all of the memory for the OM_DISPOSEd object.

1.8 OM_SET

This method is used to set an object's attributes. The Intuition function SetAttr() calls this method. It receives the following arguments:

```
struct opSet
{
    ULONG MethodID;
    struct TagItem *ops_AttrList;
    struct GadgetInfo *ops_GInfo;
};
```

For the OM_SET method, the new class's dispatcher should process the attributes your class recognizes and have the superclass process any unrecognized attributes. Note that a subclass dispatcher can directly process attributes it inherits from a superclass, rather than passing the message on to the superclass.

Note that `mytextlabelclass` treats the `OM_UPDATE` method exactly like the `OM_SET` method. This is OK because these two methods are functionally equivalent for `imageclass` classes.

1.9 OM_GET

Retrieve an object's attribute. This method receives the following parameters:

```
struct opGet
{
    ULONG MethodID;
    ULONG opg_AttrID;
    ULONG *opg_Storage;
};
```

If the new class recognizes the attribute, the new class should fill in `opg_Storage`'s target with the attribute's value. If the attribute is actually the attribute of some component object, you might want to pass the message on and let the component object process the `OM_GET`. If completely unrecognized, you should pass the message to your superclass.

1.10 Imageclass Methods

`Imageclass` defines several methods of its own which subclasses of `imageclass` either have to implement or pass on to their superclass. The method IDs for `imageclass` are defined in `<intuition/imageclass.h>`. Each method requires some parameters. The `MethodID` is the only parameter common to each method.

<code>IM_DRAW</code>	<code>IM_HITTEST</code>	<code>IM_ERASEFRAME</code>	<code>IM_MOVE</code>
<code>IM_ERASE</code>	<code>IM_DRAWFRAME</code>	<code>IM_HITFRAME</code>	<code>IM_FRAMEBOX</code>

1.11 IM_DRAW

The `IM_DRAW` method is used to tell the image to render itself. The Intuition function `DrawImageState()` uses this method. `IM_DRAW` receives the following parameters:

```
struct impDraw
{
    ULONG MethodID;
    struct RastPort *imp_RPort;
    struct
    {
        WORD X;
        WORD Y;
    } imp_Offset;
```

```

    ULONG imp_State;
    struct DrawInfo *imp_DrInfo;
};

```

The `imp_State` field contains the visual state to render the image. The visual states (defined in `<intuition/imageclass.h>`) are:

<code>IDS_NORMAL</code>	idle state
<code>IDS_SELECTED</code>	for selected gadgets.
<code>IDS_DISABLED</code>	for disabled gadgets.
<code>IDS_BUSY</code>	for future functionality
<code>IDS_INDETERMINATE</code>	for future functionality
<code>IDS_INACTIVENORMAL</code>	normal, in inactive window border.
<code>IDS_INACTIVSESELECTED</code>	selected, in inactive border.
<code>IDS_INACTIVEDISABLED</code>	disabled, in inactive border.

When setting the pens to render an image, use the values from the `imp_DrInfo->dri_Pens` pen array (Note that it is possible that `imp_DrInfo` will be `NULL`). The possible pen values are defined in `<intuition/screens.h>`.

The following code fragment shows how to use the shadow color for rendering.

```

UWORD *pens = (imp->imp_DrInfo) ? imp->imp_DrInfo->dri_Pens : NULL;

if (pens)
{
    SetAPen (imp->imp_RPort, pens[SHADOWPEN]);
}

```

1.12 IM_ERASE

The `IM_ERASE` method tells an image to erase itself. The Intuition function `EraseImage()` uses this method. `IM_ERASE` receives the following parameters:

```

struct impErase
{
    ULONG MethodID;
    struct RastPort *imp_RPort;
    struct
    {
        WORD X;
        WORD Y;
    } imp_Offset;
};

```

The `mytextlabelclass` example doesn't know anything about this method, so it blindly passes this message on to the superclass. The superclass, `imageclass`, will call the graphics.library function `EraseRect()` using the dimensions found in the `imageclass` object's `Image` structure.

1.13 IM_HITTEST

IM_HITTEST returns true if a point is within the image. The Intuition function `PointInImage()` uses this method. IM_HITTEST requires the following parameters:

```
struct impHitTest
{
    ULONG MethodID;
    struct
    {
        WORD X;
        WORD Y;
    } imp_Point;
};
```

The `mytextlabelclass` example blindly passes this method on to its superclass. The superclass, `imageclass`, will return TRUE if the point is within the old Image structure box.

1.14 IM_DRAWFRAME

The IM_DRAWFRAME method instructs the image to render itself within the confines of the given rectangle. It receives the following parameters:

```
struct impDraw
{
    ULONG MethodID;
    struct RastPort *imp_RPort;
    struct
    {
        WORD X;
        WORD Y;
    } imp_Offset;

    ULONG imp_State;
    struct DrawInfo *imp_DrInfo;

    struct
    {
        WORD Width;
        WORD Height;
    } imp_Dimensions;
};
```

The Width and Height fields provide the object's rectangular bounds. How the image object deals with the frame is implementation specific. Typically, a scaleable image will scale itself as best it can to fit into the rectangle. The `mytextlabelclass.c` example does not actually implement this method, instead `mytextlabelclass` treats IM_DRAWFRAME like the IM_DRAW method.

In general, applications that use this method to draw an object should use the IM_ERASEFRAME method (see below) to erase it. This will ensure that

the image was erased at the proper scale.

1.15 IM_ERASEFRAME

The IM_ERASEFRAME method instructs an image confined to a given rectangle to erase itself. Normally this method is used to erase an image drawn using the IM_DRAWFRAME method. This method expects the following parameters:

```
struct impErase
{
    ULONG MethodID;
    struct RastPort *imp_RPort;
    struct
    {
        WORD X;
        WORD Y;
    } imp_Offset;

    /* these parameters only valid for IM_ERASEFRAME */
    struct
    {
        WORD Width;
        WORD Height;
    } imp_Dimensions;
};
```

The mytextlabelclass example blindly passes this method on to its superclass. The superclass treats IM_ERASEFRAME just like IM_ERASE.

1.16 IM_HITFRAME

The IM_HITFRAME method is used to determine if a point is within an image that is contained within (or scaled to) the given rectangle. This method is intended to test images that were rendered using IM_DRAWFRAME. This method receives the following parameters:

```
struct impHitTest
{
    ULONG MethodID;
    struct
    {
        WORD X;
        WORD Y;
    } imp_Point;

    struct
    {
        WORD Width;
        WORD Height;
    } imp_Dimensions;
};
```

The mytextlabelclass example blindly passes this method on to its superclass. The superclass treat this meothd just like the IM_HITTEST method.

1.17 IM_MOVE

The IM_MOVE method erases and redraws an image. It is intended for use in some subclass of imageclass that performs animation or smooth dragging. Currently, no public boopsi classes implement this method.

1.18 IM_FRAMEBOX

The IM_FRAMEBOX method returns size information for an image (usually some sort of frame image). The following parameters are associated with the IM_FRAMEBOX method.

```
struct impFrameBox
{
    ULONG MethodID;
    struct IBox *imp_ContentsBox; /* Application supplied IBox for the result ↵
    */
    struct IBox *imp_FrameBox;    /* Rectangle to frame */
    struct DrawInfo *imp_DrInfo;
    ULONG imp_FrameFlags;
};
```

This method is used to ask the image what size it would like to be, if it had to frame the rectangle in the imp_FrameBox field. This method normally applies only to image classes that put a frame around some object (like frameiclass). By passing the dimensions and position of a rectangle, the framing image determines the position and size it should be to properly "frame" the object bounded by the imp_FrameBox rectangle. IM_FRAMEBOX stores the result in the IBox structure pointed to by imp_ContentsBox. This method allows an application to use a framing image without having to worry about image specific details such as accounting for the thickness of the frame or centering the frame around the object.

The imp_FrameFlags field is a bit field used to specify certain options for the IM_FRAMEBOX method. Currently, there is only one defined for it, FRAMEF_SPECIFY. If this bit is set, IM_FRAMEBOX has to use the width and height supplied to it in the imp_FrameBox field (even if these are too small!) as the frame dimensions. It can only adjust its position, typically to center the object as best as possible.

This method is not supported by the mytextlabelclass example. It passes this message to its superclass which does not support this method either. When the message returns from the superclass, the return value will be zero, indicating to the application that this method is not supported.

1.19 Image Class Example

The image class example code, `mytextlabelclass.c`, illustrates a complete custom image class. This image class provides an application with textual labels that have a particular character underlined. This is useful for indicating which key controls a gadget (although the example provided only utilizes `imageclass` objects; there are no gadgets involved).

A custom image can be used in the place of any standard Intuition Image structure. For example, an application can attach an `imageclass` object to: the `GadgetRender` and `SelectRender` fields of a `Gadget` structure (defined in `<intuition/intuition.h>`), the `ReqImage` field of a `Requester` structure, or even the `ItemFill` field of the `MenuItem` structure.

Under Intuition V36, the `DrawImage()` function passed an invalid `DrawInfo` structure, therefore it wasn't possible to use a custom `imageclass` object and the `DrawImage()` together. With V37, a `NULL` `DrawInfo` is passed when no valid `DrawInfo` is available.

The example code (`usemyIC.c`) initializes and uses a custom `imageclass` object. Notice that `usemyIC.c` directly manipulates fields within the `Image` structure embedded within the `boopsi` `imageclass` object. This is legal for image classes whose immediate superclass is `imageclass` (for the `LeftEdge`, `TopEdge`, `Width`, `Height`, `ImageData`, `PlanePick`, and `PlaneOnOff` `Image` structure fields only; the other `Image` structure fields are off limits). Indirect subclasses of `imageclass` may not alter the values in the embedded `Image` structure as future direct subclasses of `imageclass` may need to know about changes to values in the `Image` structure.