# Libraries

**COLLABORATORS**

| | *TITLE* :<br><br>Libraries | |
|---|---|---|

| *ACTION* | *NAME* | *DATE* | *SIGNATURE* |
|---|---|---|---|
| WRITTEN BY | | March 28, 2025 | |

**REVISION HISTORY**

| NUMBER | DATE | DESCRIPTION | NAME |
|---|---|---|---|
| | | | |

# Contents

# Chapter 1

# Libraries

## 1.1 Amiga® RKM Libraries: 12 Boopsi--Object Oriented Intuition

Boopsi is an acronym for Basic Object Oriented Programming System for Intuition. Using the Object Oriented Programming (OOP) model, Boopsi represents certain Intuition entities, like Gadgets and Images, as objects.

There are many advantages to using Boopsi:

* Boopsi makes Intuition customizable and extensible. Boopsi programmers can create new types of Boopsi objects to suit the needs of their applications. These new types of objects are part of Intuition and can be made public so other applications can use them. Because applications can share the new types, application writers don't have to waste their time duplicating each other's efforts writing the same objects.

* New types of Boopsi objects can build on old types of Boopsi objects, inheriting the old object's behavior. The result is that Boopsi programmers don't have to waste their time building new objects from scratch, they simply add to the existing object.

* OOP and Boopsi apply the concept of interchangeable parts to Intuition programming. A Boopsi programmer can combine different Boopsi objects (like gadgets and images) to create an entire Graphical User Interface (GUI). The Boopsi programmer doesn't have take the time to understand or implement the inner workings of these objects. The Boopsi programmer only needs to know how to interact with Boopsi objects and how to make them interact with each other.

* Boopsi objects have a consistent, command-driven interface. To the Boopsi programmer, there is no difference between displaying a text, border, or bitmap-based Boopsi image, even though they are rendered quite differently. Each image object accepts a single command to tell it to render itself.

Before reading this chapter, you should already be familiar with several Amiga concepts. Boopsi is built on top of Intuition and uses many of its structures. These include Intuition gadgets, images, and windows. Boopsi also uses the tag concept to pass parameters. The "Utility Library" chapter of this manual discusses tags. The "Utility Library" chapter also

discusses callback Hooks, which are important to the later sections of this chapter.



## 1.2 12 Boopsi--Object Oriented Intuition / OOP Overview

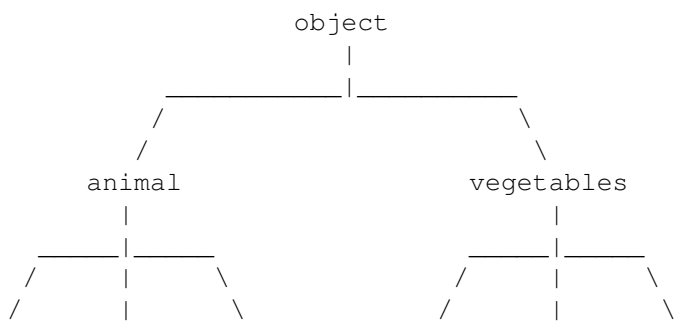
Understanding Boopsi requires an understanding of several of the concepts behind Object Oriented Programming. This section is a general overview of these concepts as they pertain to Boopsi. Because Boopsi is in part based on the concepts present in the OOP language Smalltalk, a reference book on Smalltalk may provide a deeper understanding of Boopsi in general. Timothy Budd's book entitled A Little Smalltalk (Addison-Wesley Publishing ISBN 0-201-10698-1) is a good start.

In the Boopsi version of the Object Oriented Programming model, everything is an Object. For example, a proportional gadget named myprop is an object. Certain objects have similar characteristics and can be classified into groups called classes. As objects, Rover the dog, Bob the cat, and Sam the bird are all distinct objects but they all have something in common, they can all be classified as animals. As objects, myprop the proportional gadget, mystring the string gadget, and mybutton the button gadget all have something in common, they can all be classified as gadgets. A specific object is an instance of a particular class ("Rover" is an instance of class "animal", "myslidergadget" is an instance of class "gadget").

Notice that, although Rover, Bob, and Sam can all be classified as animals, each belongs to a subgroup of the animal class. "Rover" is an instance of class "dog", "Bob" is an instance of class "cat", and "Sam" is an instance of class "bird". Because each of these animal types share common characteristics, each type makes up its own class. Because dog, cat, and bird are subclassifications of the animal class, they are known as subclasses of the animal class. Conversely, the animal class is the superclass of the dog, cat, and bird classes.

Following the branches upward from class to superclass will bring you to a universal root category from which all objects are derived. The OOP language Smalltalk calls this class "Object".

Figure 12-1: Object Diagram



dog cat mouse peas corn spinach

Like Smalltalk, Boopsi also has a universal root catagory, rootclass. Currently, Intuition defines three immediate subclasses of rootclass. The first, gadgetclass, is the class of Boopsi gadgets. The second class, imageclass, makes up the class of Boopsi images.

Unlike gadgetclass and imageclass, the remaining subclass, icclass, does not correspond to an existing Intuition entity, it is a concept new to Intuition. Icclass, or interconnection class, allows one Boopsi object to notify another Boopsi object when a specific event occurs. For example, consider a Boopsi proportional gadget and a Boopsi image object that displays an integer value. An application can connect these two objects so that the prop gadget tells the image object the prop gadget's current value, which the image object displays. Every time the user slides the prop gadget, the prop gadget notifies the image of the change and the image updates its display to reflect the prop gadget's current integer value. Because these objects are talking to each other rather than the application, the updates happen automatically. The application doesn't have to talk to the two objects, it only has to connect them.

Figure 12-2: Simple Boopsi Diagram

An object's characteristics and behavior are determined by its class. Each class can define a set of attributes and a set of methods that apply to all objects of that class. An attribute is a variable characteristic of an object. For example, an attribute for the animal class could be the number of legs an animal object has. An example of a Boopsi attribute is the X coordinate of a Boopsi image object. The data that makes up the values of an object's attributes is collectively known as the instance data for that object.

The behavior of an object depends upon the set of methods associated to it by its class. A method is basically a function that applies to objects of that class. An example of a Boopsi method is the imageclass method IM_DRAW. This method tells a Boopsi image to draw itself. All Boopsi actions are carried out via methods.

From the Object Diagram, two of the methods of the "animal" class could be "eat" and "sleep". One of the methods of the "dog" class could be "bark". Notice that instances of the "dog" class can do more than just bark, they can also eat and sleep. This is because a subclass inherits methods from its superclasses. If there were a subclass of dog called "attack dog", all instances of that class would be able to bark, eat, and sleep, as well as "attack". Due to inheritance, a subclass has all of the methods and all of the attributes of its superclass. For example, the IA_Height attribute is defined by imageclass. All instances of the subclasses of imageclass have their own IA_Height attribute, even though the subclasses do not explicitly define IA_Height. In turn, all instances of subclasses of the imageclass subclasses also inherit the IA_Height attribute. All classes on levels below a class will inherit its methods and attributes.

When an application or a Boopsi object wants another Boopsi object to perform a method, it passes it a command in the form of a Boopsi message. A Boopsi message tells an object which method to perform. The message may also contain some parameters that the method requires.

```
    Watch Out!
    ----------
    The term "message" used in object oriented terminology can be little
    confusing to the Amiga programmer because the Boopsi message has
    nothing to do with an Exec message.
```

Boopsi classes can be either public or private. Public classes have ASCII names associated with them and are accessible to all applications. Private classes have no ASCII name and normally can only be accessed by the application that created the private class.

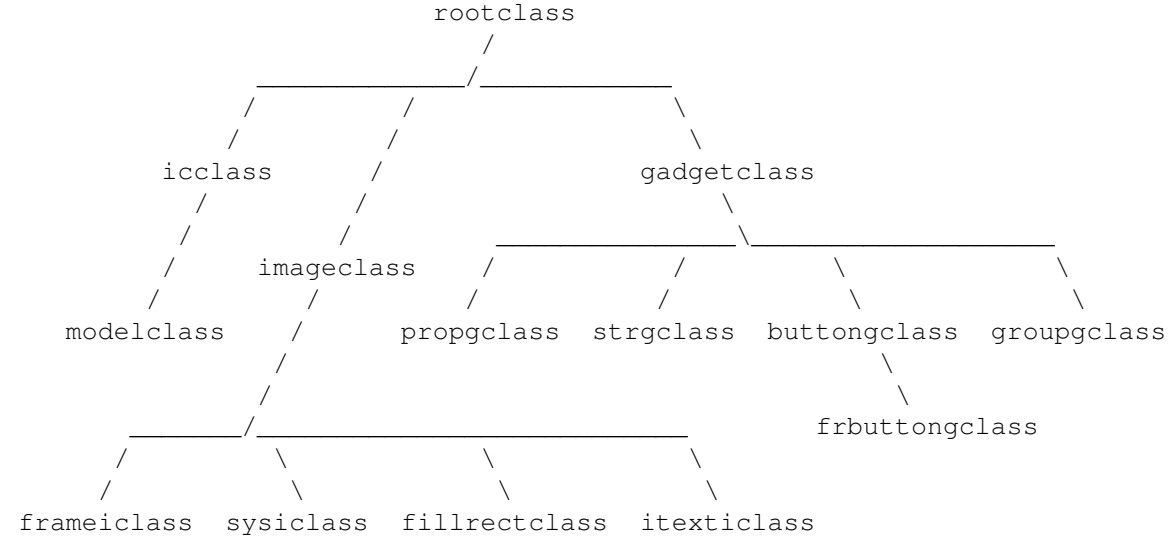


## 1.3 12 / OOP Overview / Using Boopsi

There are several levels on which a programmer can use Boopsi. The most elementary level is to use Intuition functions to create and manipulate Boopsi objects that are instances of existing, public classes.

At present there is a hierarchy of 14 public classes built into Intuition:

Figure 12-3: Class Diagram



What About the Boopsi Messages and Methods?

## 1.4 12 / / Using Boopsi / Boopsi and Tags

Boopsi uses tag lists to pass and manipulate its attributes. To Boopsi, each TagItem (defined in <utility/tagitem.h>) in a tag list is an attribute/value pair. The TagItem.ti_Tag field contains an ID for the attribute and the ti_Data field holds the attribute's value.

For example, the string gadget class defines an attribute called STRINGA_LongVal, which is the current integer value of the gadget. Certain gadgetclass objects have an attribute called GA_Image. Its value is not an integer, it is a pointer to an image.

Note that these tag lists can also contain utility.library Global System control tags (like TAG_SKIP and TAG_DONE), which Boopsi uses in processing its tag lists. Any application that ends up processing these lists should do so using the tag manipulation functions from utility.library. For more information on tags and utility.library, see the "Utility Library" chapter of this manual.

## 1.5 12 / / Using Boopsi / Creating an Object

The Intuition function NewObjectA() creates a Boopsi object:

```
    mynewobject = APTR NewObjectA(Class *privclass, UBYTE *pubclass,
                                  struct TagItem *myattrs)
```

The pointer that NewObjectA() returns is a pointer to a Boopsi object. In general, Boopsi objects are "black boxes". This means the inner workings of Boopsi objects are not visible to the application programmer, so the programmer does not know what goes on inside it. This really means the inner workings of these objects are none of your business. Unless otherwise documented, only use an object pointer as a handle to the object.

To create an object, NewObjectA() needs to know what class the new object is an instance of. To create a public class object, pass a NULL pointer in privclass and an ASCII string in pubclass naming the object's public class. The privclass pointer is used to create a private class object, which is covered in the "Creating a Boopsi Class" section later in this chapter.

The myattrs tag list is a list of tag/value pairs, each of which contains an initial value for some object attribute. Most objects have a set of attributes associated with them, so each attribute has a tag name. For Boopsi gadgets and images, the attributes include some of the values from the old Gadget and Image structures (position, size, etc.).

Most applications use the stack-based version of NewObjectA(), NewObject(), to create objects. This allows an application to build the tag list of object attributes on the stack rather than having to allocate and initialize a tag list. A code sample from a program that creates a

Boopsi string gadget might look like this:

```
    mystringgadget = (struct Gadget *)NewObject(NULL, "strgclass",
                                                GA_ID,           1L,
                                                GA_Left,         0L,
                                                GA_Top,          0L,
                                                STRINGA_LongVal, 100L,
                                                TAG_END);
```

If NewObject() is successful, it returns a pointer to a new Boopsi gadget object. Otherwise, it returns NULL. The class "strgclass" is one of the public classes built into Release 2. It is a class of string gadgets.

If you look at the diagram of the public classes built into Intuition, you'll see that strgclass is a subclass of gadgetclass. In the example above, the attribute tag IDs that start with "GA_" are defined by gadgetclass and not by strgclass. This is because strgclass inherits these attributes from its superclass, gadgetclass. The other attribute, STRINGA_LongVal, is defined by strgclass. It does two things. First, it tells the object that it is a special type of string gadget which only handles an integer value rather than a generic ASCII string. Second, it passes the object its initial integer value.

## 1.6 12 / / Using Boopsi / Disposing of an Object

When an application is done with an object it has to dispose of the object. To dispose of an object, use the Intuition function DisposeObject():

```
    VOID DisposeObject(APTR boopsiobject);
```

where boopsiobject is a pointer to the Boopsi object to be disposed. Note that some classes allow applications to connect child objects to a parent object so that when the application deletes the parent object, it automatically disposes of all of its children. Be careful not to dispose of an object that has already been disposed.

## 1.7 12 / / Using Boopsi / Setting an Existing Object's Attributes

An object's attributes are not necessarily static. An application can ask an object to set certain object attributes using the SetAttrs() function:

```
    ULONG SetAttrs(APTR myobject, Tag1, Value1, ...);
```

Because Boopsi gadgets require some extra information about their display, they use a special version of this function, SetGadgetAttrs():

```
    ULONG SetGadgetAttrs(struct Gadget *myobject, struct Window *w,
                         struct Requester *r, Tag1, Value1, ...);
```

Here myobject is a pointer to the Boopsi object, w points to the gadget's window, r points to the gadget's requester, and the tag/value pairs are

the attributes and their new values. The return value of SetAttrs() and SetGadgetAttrs() is class specific. In general, if the attribute change causes a visual change to some object, the SetAttrs()/SetGadgetAttrs() function should return a non-zero value, otherwise, these functions should return zero (see the Boopsi Class Reference in "Appendix B" of this manual for information on the return values for specific classes). The following is an example of how to set the current integer value and gadget ID of the gadget created in the NewObject() call above:

```
SetGadgetAttrs(mystringgadget, mywindow, NULL, STRINGA_LongVal,   75L,
                                               GA_ID,             2L,
                                               TAG_END));
```

This changes two of mystringgadget's attributes. It changes the gadget's current integer value to 75 and it changes the gadget's ID number to 2.

Note that it is not OK to call SetGadgetAttrs() on a Boopsi object that isn't a gadget, nor is it OK to call SetAttrs() on a Boopsi gadget.

Not all object attributes can be set with SetGadgetAttrs()/SetAttrs(). Some classes are set up so that applications cannot change certain attributes. For example, the imagery for the knob of a proportional gadget cannot be altered after the object has been created. Whether or not a specific attribute is "settable" is class dependent. For more information about the attributes of specific classes, see the Boopsi Class Reference in the Appendix B of this manual.

## 1.8 12 / / Using Boopsi / Getting an Object's Attributes

The Intuition function GetAttr() asks an object what the value of a specific attribute is:

```
ULONG GetAttr(ULONG attrID, APTR myobject, ULONG *mydata);
```

where attrID is the attribute's ID number, myobject is the object to get the attribute from, and mydata points to a data area that will hold the attribute value. This function returns a 0L if the object doesn't recognize the attribute, otherwise it returns some non-zero value, the meaning of which depends on the class. In most cases, GetAttr() returns a 1 when it is successful.

Not all object attributes are obtainable using the GetAttr() function. Some classes are set up so that applications cannot query the state of certain attributes. For example, using the GA_Image attribute, an application can give a Boopsi prop gadget (propgclass) an Image structure which the gadget uses as the imagery for its knob. This attribute is not "gettable" as there is no need for an application to have to ask the gadget for the structure that the application passed it in the first place. Whether or not a specific attribute is "gettable" is class dependent. For more information about the attributes of specific classes, see the Boopsi Class Reference in the Appendix B of this manual.

## 1.9 12 / / Using Boopsi / What About the Boopsi Messages and Methods?

According to the "OOP Overview" section, for an object to perform a method, something has to pass it a Boopsi message. The previous section discussed using Intuition functions to ask an object to do things like set and get attributes. The functions in the previous section seem to completely ignore all that material about methods and messages. What happened to the methods and messages?

Nothing--these functions don't ignore the OOP constructs, they just shield the programmer from them. Each of these functions corresponds to a Boopsi method:

| | |
|---|---|
| NewObject() | OM_NEW |
| DisposeObject() | OM_DISPOSE |
| SetAttrs()/SetGadgetAttrs() | OM_SET |
| GetAttr() | OM_GET |

These methods are defined on the rootclass level, so all Boopsi classes inherit them. The Intuition functions that correspond to these methods take care of constructing and sending a Boopsi message with the appropriate method ID and parameters.

## 1.10 12 / OOPOverview / The Public Classes

Intuition contains 14 public classes, all of which are descendants of the rootclass. There are three primary classes that descend directly from rootclass: imageclass, gadgetclass, and icclass.

The Imageclass Subclasses  The Gadgetclass Subclasses

## 1.11 12 / / The Public Classes / The Imageclass Subclasses

Normally, an application does not create an imageclass object. Instead, it will use a subclass of imageclass. Currently, there are four subclasses: frameiclass, sysiclass, fillrectclass, and itexticlass.

frameiclass
: An embossed or recessed rectangular frame image, that renders itself using the proper DrawInfo pens. This class is intelligent enough to bound or center its contents.

sysiclass
: The class of system images. The class includes the images for the system and GadTools gadgets.

fillrectclass
: A class of rectangle images that have frame and patternfill support.

itexticlass
: A specialized image class used for rendering text.

For more information on these classes see the Boopsi Class Reference in the Appendix B of this manual. It describes all of the existing public classes, their methods, and their attributes.

The Gadgetclass Subclasses

## 1.12 12 / / The Public Classes / The Gadgetclass Subclasses

Like imageclass, applications do not normally create objects of gadgetclass, but instead create objects of its subclasses. Currently, gadgetclass has four subclasses:

propgclass
    An easy to implement, horizontal or vertical proportional gadget.

strgclass
    A string gadget.

groupgclass
    A special gadget class that creates one composite gadget out of several others.

buttongclass
    A button gadget that keeps sending button presses while the user holds it down.

buttongclass has a subclass of its own:

frbuttonclass
    A buttongclass gadget that outlines its imagery with a frame.

For specific information on these classes, see the Boopsi Class Reference in the Appendix B of this manual.

## 1.13 12 / OOP Overview / Making Gadget Objects Talk to Each Other

One use for a proportional gadget is to let the user change some integer value, like the red, green, and blue components of a color. This type of prop gadget is commonly accompanied by an integer string gadget, enabling the user to adjust one integer value by either typing the value into the string gadget or by scrolling the prop gadget. Because these two gadgets reflect the value of the same integer, when the user adjusts the state of one of the gadgets (and thus changing the integer value), the other gadget should automatically update to reflect the new integer value.

When the user manipulates a conventional gadget, the gadget sends messages to an IDCMP port to indicate the state change (for information on IDCMP, see the "Intuition Input and Output Methods" chapter of this manual). To connect the string and prop gadgets from the previous paragraph, an application would have to listen for the IDCMP messages from two different gadgets, interpret the IDCMP message's meaning, and manually update the gadgets accordingly. Essentially, the application is responsible for

"gluing" the gadgets together. This unnecessarily complicates an application, especially when that application already has to listen for and interpret many other events.

Boopsi gadgets simplify this. By setting the appropriate attributes, an application can ask a Boopsi gadget to tell some other object when its state changes. One of the attributes defined by gadgetclass is ICA_TARGET (defined in <intuition/icclass.h>). The ICA_TARGET attribute points to another Boopsi object. When certain attributes in a Boopsi gadget change (like the integer value of a prop gadget), that gadget looks to see if it has an ICA_TARGET. If it does, it sends the target a message telling it to perform an OM_UPDATE method.

The OM_UPDATE method is defined by rootclass. This is basically a special type of OM_SET method that is used specifically to tell a Boopsi object that another Boopsi object's state changed. Only Boopsi objects send OM_UPDATE messages. Note that standard classes of Boopsi gadgets only send out OM_UPDATE messages as a result of the user changing the state of the gadget (scrolling the prop gadget, typing a new number into an integer gadget, etc.). These gadgets do not send out OM_UPDATE messages when they receive OM_SET or OM_UPDATE messages.

A Boopsi propgclass object has only one attribute that triggers it to send an OM_UPDATE request: PGA_Top. This attribute contains the integer value of the prop gadget. Every time the user moves a prop gadget, the PGA_Top attribute changes. If the prop gadget has an ICA_TARGET, the prop gadget will tell the target object that the PGA_Top value has changed.

A Boopsi integer string gadget (a strgclass object) also has only one attribute that triggers it to send an OM_UPDATE request: STRINGA_LongVal. value contains the integer value of the integer string gadget. Like the prop gadget, if the integer string gadget has an ICA_TARGET, when the user changes the gadget's integer value (STRINGA_LongVal), the string gadget will tell the target object that the STRINGA_LongVal value has changed.

When a Boopsi gadget sends an OM_UPDATE message, it passes the ID of the attribute that changed plus that attribute's new value. For example, if the user typed a 25 into a Boopsi integer string gadget, that gadget would send an OM_UPDATE message to its ICA_TARGET saying in essence, "Hey, STRINGA_LongVal is 25".

If this string gadget's ICA_TARGET is a propgclass object, the propgclass object will become confused because it has no idea what a STRINGA_LongVal attribute is. The string gadget needs to map its STRINGA_LongVal ID to the PGA_Top ID. This is what the ICA_MAP attribute is for.

The ICA_MAP attribute is defined by gadgetclass (it is also defined for icclass--more on that later). It accepts a tag list of attribute mappings. When a gadget sends out an OM_UPDATE message, it uses this map to translate a specific attribute ID to another attribute ID, without changing the value of the attribute. Each TagItem in the ICA_MAP makes up a single attribute mapping. The TagItem.ti_Tag of the mapping is the ID of an attribute to translate. The gadget translates that attribute ID to the attribute ID in TagItem.ti_Data. For example, an ICA_MAP that maps a string gadget's STRINGA_LongVal attribute to a prop gadget's PGA_Top attribute looks like this:

```
    struct TagItem slidertostring[] = {
        {PGA_Top, STRINGA_LongVal},
        {TAG_END, }
    };
```

Note that it is OK to have an ICA_TARGET without having an ICA_MAP. In cases where a gadget and its ICA_TARGET have a set of attributes in common, it would be unnecessary to use an ICA_MAP to match a gadget's attributes, as they already match.

The following example, Talk2boopsi.c, creates a prop gadget and an integer string gadget which update each other without the example program having to process any messages from them.

Talk2boopsi.c

## 1.14 12 / OOP Overview / Making Gadgets Talk to an Application

There are two questions that the example above brings to mind. The first is, "What happens if the user types a value into the string gadget that is beyond the bounds of the prop gadget?" The answer is simple: very little. The prop gadget is smart enough to make sure its integer value does not go beyond the bounds of its display. In the example, the prop gadget can only have values from 0 to 90. If the user tries to type a value greater than 90, the prop gadget will set itself to its maximum of 90. Because the integer string gadget doesn't have any bounds checking built into it, the example needs to find an alternative way to check the bounds.

The other question is, "How does talk2boopsi.c know the current value of the gadgets?" That answer is simple too: it doesn't. The example doesn't ask the gadgets what their current values are (which it would do using GetAttr()) and the example doesn't pay attention to gadget events at the window's IDCMP port, so it isn't going to hear about them.

One easy way to hear about changes to the gadget events is to listen for a "release verify". Conventional Intuition gadgets can trigger a release verify IDCMP event when the user finishes manipulating the gadget. Boopsi gadgets can do this, too, while continuing to update each other.

To make Talk2boopsi.c do this would require only a few changes. First, the window's IDCMP port has to be set up to listen for IDCMP_GADGETUP events. Next, the example needs to set the gadget's GACT_RELVERIFY flags. It can do this by setting the gadgetclass GA_RelVerify attribute to TRUE for both gadgets. That's enough to trigger the release verify message, so all Talk2boopsi.c needs to do is account for the new type of IDCMP message, IDCMP_GADGETUP. When Talk2boopsi.c gets a release verify message, it can use GetAttr() to ask the integer gadget its value. If this value is out of range, it should explicitly set the value of the integer gadget to a more suitable value using SetGadgetAttrs().

Using the GACT_RELVERIFY scheme above, an application will only hear about changes to the gadgets after the user is finished changing them. The application does not hear all of the interim updates that, for example, a prop gadget generates. This is useful if an application only needs to

hear the final value and not the interim update.

It is also possible to make the IDCMP port of a Boopsi gadget's window the ICA_TARGET of the gadget. There is a special value for ICA_TARGET called ICTARGET_IDCMP (defined in <intuition/icclass.h>). This tells the gadget to send an IDCMP_IDCMPUPDATE class IntuiMessage to its window's IDCMP port. Of course, the window has to be set up to listen for IDCMP_IDCMPUPDATE IntuiMessages. The Boopsi gadget passes an address in the IntuiMessage.IAddress field. It points to an attribute tag list containing the attribute (and its new value) that triggered the IDCMP_IDCMPUPDATE message. An application can use the utility.library tag functions to access the gadget's attributes in this list. Using this scheme, an application will hear all of the interim gadget updates. If the application is using a gadget that generates a lot of interim OM_UPDATE messages (like a prop gadget), the application should be prepared to handle a lot of messages.

Using this IDCMP_IDCMPUPDATE scheme, if the gadget uses an ICA_MAP to map the attribute to a special dummy attribute ICSPECIAL_CODE (defined in <intuition/icclass.h>), the IntuiMessage.Code field will contain the value of the attribute. Because the attribute's value is a 32-bit quantity and the IntuiMessage.Code field is only 16 bits wide, only the least significant 16 bits of the attribute will appear in the IntuiMessage.Code field, so it can't hold a 32-bit quantity, like a pointer. Applications should only use the lower 16 bits of the attribute value.

## 1.15 12 / OOP Overview / The Interconnection Classes

The IDCMP_IDCMPUPDATE scheme presents a problem to an application that wants to make gadgets talk to each other and talk to the application. Boopsi gadgets only have one ICA_TARGET. One Boopsi gadget can talk to either another Boopsi object or its window's IDCMP port, but not both. Using this scheme alone would force the application to update the integer value of the gadgets, which is what we are trying to avoid in the first place.

One of the standard Boopsi classes, icclass, is a class of information forwarders. An icclass object receives OM_UPDATE messages from one object and passes those messages on to its own ICA_TARGET. If it needs to map any incoming attributes, it can use its own ICA_MAP to do so.

Icclass has a subclass called modelclass. Using a modelclass object, an application can chain a series of these objects together to set up a "broadcast list" of icclass objects. The modelclass object is similar to the icclass object in that it has its own ICA_TARGET and ICA_MAP. It differs in that an application can use the modelclass OM_ADDMEMBER method to add icclass objects to the modelclass object's broadcast list.

The OM_ADDMEMBER method is defined by rootclass. It adds one Boopsi object to the personal list of another Boopsi object. It is up to the Boopsi object's class to determine the purpose of the objects in the list. Unlike the other methods mentioned so far in this chapter, OM_ADDMEMBER does not have an Intuition function equivalent. To pass an OM_ADDMEMBER message to an object use the amiga.lib function DoMethodA(), or its stack-based equivalent, DoMethod():

```
    ULONG DoMethodA(Object *myobject, Msg boopsimessage);
    ULONG DoMethod(Object *myobject, ULONG methodID, ...);
```

The return value is class-dependent. The first argument to both of these functions points to the object that will receive the Boopsi message.

For DoMethodA(), boopsimessage is the actual Boopsi message. The layout of it depends on the method. Every method's message starts off with an Msg (from <intuition/classusr.h>):

```
    typedef struct {
        ULONG MethodID; /* Method-specific data may follow this field */
    } *Msg;
```

The message that the OM_ADDMEMBER method uses looks like this (from <intuition/classusr.h>):

```
    struct opMember {
        ULONG    MethodID;
        Object   *opam_Object;
    };
```

where MethodID is OM_ADDMEMBER and opam_Object points to the object to add to myobject's list.

DoMethod() uses the stack to build a message. To use DoMethod(), just pass the elements of the method's message structure as arguments to DoMethod() in the order that they appear in the structure. For example, to ask the Boopsi object myobject to add the object addobject to its personal list:

```
    DoMethod(myobject, OM_ADDMEMBER, addobject);
```

To rearrange Talk2boopsi.c so that it uses a modelclass object (also known as a model):

* Create the integer and prop gadget.
* Create the model.
* Create two icclass objects, one called int2prop and the other called prop2int.
* Make the model the ICA_TARGET of both the integer gadget and the prop gadget. The gadgets do not need an ICA_MAP.
* Using DoMethod() to call OM_ADDMEMBER, add the icclass objects to the model's personal list.
* Make the prop gadget the ICA_TARGET of int2prop. Make the integer gadget the ICA_TARGET of prop2int.
* Create an ICA_MAP map list for int2prop that maps STRINGA_LongVal to PGA_Top. Create an ICA_MAP map list for prop2int that maps PGA_Top to STRINGA_LongVal. Make the ICA_TARGET of the model ICTARGET_IDCMP.

Diagrammatically, the new Talk2boopsi.c should look something like this:

Figure 12-4: ICC Diagram

When either of these gadgets has some interim state change (caused by the user manipulating the gadgets), it sends an OM_UPDATE message to its ICA_TARGET, which in this case is the modelclass object. When this model gets the message, it does two things. It sends an IDCMP_IDCMPUPDATE to the IDCMP port of the gadget's window and it also sends OM_UPDATE messages to all of the objects in its personal list. When int2prop gets an OM_UPDATE message, it forwards that message to its ICA_TARGET, the prop gadget. Similarly, when prop2int gets an OM_UPDATE message, it forwards that message to its ICA_TARGET, the integer gadget.

Although in this case it isn't a problem, icclass and modelclass objects contain loop inhibition capabilities. If an icclass object (or modelclass object) receives an OM_UPDATE message, it forwards the message to its target. If somehow that forwarded message gets forwarded (or broadcast) back to the icclass object, the icclass object ignores the message. This prevents the possibility of an infinite OM_UPDATE loop.

## 1.16 12 Boopsi--Object Oriented Intuition / Creating a Boopsi Class

So far this chapter has only hinted at what is possible with Boopsi. Its power lies in its extensibility. Boopsi grants the application programmer the power to add custom features to existing classes. If an existing class comes close to your needs, you can build on that class so it does exactly what you want. If you want a class that is unlike an existing class, you can create it.

The heart of a Boopsi class is its method Dispatcher function. According to the OOP metaphor, when an application wants a Boopsi object to perform a method, it sends the object a message. In reality, that object is only a data structure, so it does not have the power to do anything. When an object receives a Boopsi message, a Boopsi message structure is passed to the dispatcher of that object's class. The dispatcher examines the message and figures out what to do about it.

For example, when an application calls SetGadgetAttrs() on an integer gadget:

```
    SetGadgetAttrs(myintegergadget, mywindow, NULL,
                   STRINGA_LongVal, 75L,
                   GA_ID,           2L,
                   TAG_END));
```

the SetGadgetAttrs() function calls the strgclass dispatcher. A Boopsi dispatcher receives three arguments: a pointer to the dispatcher's Class (defined in <intuition/classes.h>), a pointer to the object that is going to perform the method, and a pointer to the Boopsi message. In this case, the SetGadgetAttrs() function builds an OM_SET message, finds the strgclass dispatcher, and "sends" the dispatcher the OM_SET message. SetGadgetAttrs() can find the dispatcher because an object contains a reference to its dispatcher.

When the dispatcher function "gets" the message, it examines the message to find out its corresponding method. In this case, the dispatcher recognizes the message as an OM_SET message and proceeds to set myintegergadget's attributes.

An OM_SET message looks like this (defined in <intuition/classusr.h>):

```
    struct opSet {
        ULONG MethodID;               /* This will be set to OM_SET      */
        struct TagItem *ops_AttrList; /* A tag list containing the       */
                                      /*     attribute/value pairs of    */
                                      /*     the attributes to set.      */
        struct GadgetInfo *ops_GInfo; /* Special information for gadgets */
    }
```

The OM_SET message contains a pointer to a tag list in ops_AttrList that looks like this:

```
    {STRINGA_LongVal, 75L},
    {GA_ID, 2L},
    {TAG_END,}
```

The strgclass dispatcher scans through this tag list and recognizes the STRINGA_LongVal attribute. The dispatcher sets myintegergadget's internal STRINGA_LongVal value to the corresponding value (75L) from the attribute/value pair.

The strgclass dispatcher continues to scan through the tag list. When it finds GA_ID, it does not process it like STRINGA_LongVal. The strgclass dispatcher's OM_SET method does not recognize the GA_ID attribute because strgclass inherited the GA_ID attribute from gadgetclass. To handle setting the GA_ID attribute, the strgclass dispatcher passes on the OM_SET message to its superclass's dispatcher. The strgclass dispatcher passes control to the gadgetclass dispatcher, which knows about the GA_ID attribute.



## 1.17 12 / Creating a Boopsi Class / Building On Existing Public Classes

A program can create its own subclasses which build on the features of existing classes. For example, a program could create a subclass of modelclass named rkmmodelclass. Rkmmodelclass builds on modelclass by adding a new attribute called RKMMOD_CurrVal. This purpose of this attribute is simply to hold an integer value.

Because this new attribute is built into an rkmmodel object, the object could be implemented so that it exercises a certain amount of control over that value. For example, rkmmodelclass could be implemented so an rkmmodel performs bounds checking on its internal value. When an application asks an rkmmodel to set its internal RKMMOD_CurrVal, the rkmmodel makes sure the new value is not beyond a maximum value. If the new value is beyond the maximum, it sets its current value to the maximum.

After the rkmmodelclass object has set its internal RKMMOD_CurrVal, it can broadcast the change on to objects in its broadcast list.

The dispatcher for rkmmodelclass does not have to do a lot of work because it inherits most of its behavior from its superclasses. The rkmmodelclass has to take care of setting aside memory for the RKMMOD_CurrVal attribute and processing any OM_SET requests to set the RKMMOD_CurrVal attribute. For any other attributes or methods, the rkmmodelclass dispatcher passes on processing to its superclass, modelclass.

Building Rkmmodelclass

## 1.18 12 / / Building On Existing Public Classes / Building Rkmmodelclass

So far, the theoretical class rkmmodelclass has just one attribute, RKMMOD_CurrVal. A couple of extra attributes can make it more useful. Because the rkmmodel object maintains an upper limit on its RKMMOD_CurrVal integer value, it would be useful if that upper limit was variable. Using a new attribute, RKMMOD_Limit, an application can tell a rkmmodel what its upper limit is. The rkmmodel will enforce the limit internally, so the application doesn't have to worry about it.

Another useful addition is a pulse increment and decrement for RKMMOD_CurrVal. Whenever the model receives an increment or decrement command, it increments or decrements its internal value. To make the example class simple, rkmmodelclass implements incrementing and decrementing by creating "dummy" attributes called RKMMOD_Up and RKMMOD_Down. When an rkmmodel receives an OM_SET message for one of these attributes, it increments or decrements RKMMOD_CurrVal. An rkmmodelclass object does not care what the value of the RKMMOD_Up and RKMMOD_Down attributes are, it only cares that it received an OM_UPDATE about it.

There are two pieces of data that make up this new class's instance data: the rkmmodel's current value (RKMMOD_CurrVal) and the upper limit of the rkmmodel (RKMMOD_Limit). The example class consolidates them into one structure:

```
struct RKMModData {
    ULONG currval;
    ULONG vallimit;
};
```

## 1.19 12 / Creating a Boopsi Class / Writing the Dispatcher

The C prototype for a Boopsi dispatcher looks like this:

```
ULONG dispatchRKMModel(Class *cl, Object *recvobject, Msg msg);
```

where cl points to the Class (defined in <intuition/classes.h>) of the dispatcher, recvobject points to the object that received the message, and msg is that Boopsi message. The format of the message varies according to the method. The default Boopsi message is an Msg (from

<intuition/classusr.h>):

```
    typedef struct {
        ULONG MethodID;
    } *Msg;
```

Boopsi methods that require parameters use custom message structures. The first field of any message structure is always the method's methodID. This makes custom messages look like an Msg. The dispatcher looks at an incoming message's first field to tell what its method is. Rkmmodelclass objects respond to several rootclass methods:

OM_NEW
    This method creates a new rkmmodelclass object. It uses an opSet structure as its Boopsi message.

OM_DISPOSE
    This method tells an object to dispose of itself. It uses an Msg as its Boopsi message.

OM_SET
    This method tells an object to set one or more of its attribute values. It uses an opSet structure as its Boopsi message.

OM_UPDATE
    This method tells an object to update one or more of its attribute values. It uses an opUpdate structure as its Boopsi message.

OM_GET
    This method tells an object to report an attribute value. It uses an opGet structure as its Boopsi message.

OM_ADDTAIL
    This method tells an object to add itself to the end of an Exec list. It uses an opAddTail structure as its Boopsi message.

OM_REMOVE
    This method tells an object to remove itself from an Exec list. It uses an Msg as its Boopsi message.

OM_ADDMEMBER
    This method tells an object to add an object to its broadcast list. It uses an opMember structure as its Boopsi message.

OM_REMMEMBER
    This method tells an object to remove an object from its broadcast list. It uses an opMember structure as its Boopsi message.

OM_NOTIFY
    This method tells an object to broadcast an attribute change to its broadcast list. It uses an opSet structure as its Boopsi message.

Of these, rkmmodelclass has to process OM_NEW, OM_SET, OM_UPDATE, and OM_GET.

OM_NEW OM_GET RKMModel.c

OM_SET/OM_UPDATE  Making the New Class

## 1.20 12 / / Writing The Dispatcher / OM_NEW

The OM_NEW method returns a pointer to a newly created Boopsi object, or NULL if it failed to create the object. This method receives the following message structure (defined in <intuition/classusr.h>):

```
/* The OM_NEW method uses the same structure as OM_SET */

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

The ops_AttrList field contains a pointer to a TagItem array of attribute/value pairs. These contain the initial values of the new object's attributes. The ops_GInfo field is always NULL for the OM_NEW method.

Unlike other methods, when a dispatcher gets an OM_NEW message, the object pointer (recvobject from the dispatchRKMModel() prototype above) does not point to an object. It doesn't make sense for recvobject to point to an object because the idea is to create a new object, not act on an existing one.

The pointer normally used to pass a Boopsi object is instead used to pass the address of the object's "true class". An object's true class is the class of which the object is an instance.

The first thing the dispatcher does when it processes an OM_NEW message is pass the OM_NEW message on to its superclass's dispatcher. It does this using the amiga.lib function DoSuperMethodA():

```
ULONG DoSuperMethodA(Class *cl, Object *trueclass, Msg msg);
```

Each dispatcher passes control to its superclass. Eventually the message will arrive at the rootclass dispatcher. The OM_NEW method in the rootclass dispatcher looks at the object's true class (trueclass from the prototype) to find out which class dispatcher is trying to create a new object. Note that trueclass is not necessarily the same as the current dispatcher's class (cl from the dispatchRKMModel() prototype above), although this would be the case if the object's true class is a subclass of the current dispatcher's class.

The rootclass dispatcher uses the true class to find out how much memory to allocate for the object's instance data. Each class keeps a record of how much memory its local instance data requires. The rootclass dispatcher also looks at each class between the true class and rootclass to find out much memory the local instance data for those classes require. The rootclass dispatcher totals the amount of local instance data memory needed by the true class and each of its superclasses and allocates that much memory.

If all goes well, the rootclass dispatcher increments a private field in the true class that keeps track of how many instances of the true class there currently are. It then returns a pointer to the newly created object and passes control back to the subclass dispatcher that called it, which is icclass in the case of rkmmodelclass. If there was a problem, the rootclass dispatcher does not increment the object count and passes back a NULL.

When the rootclass dispatcher returns, the icclass dispatcher regains control from DoSuperMethodA(). DoSuperMethodA() will return either a pointer to the new object or else it returns NULL if there was an error. Although the rootclass dispatcher allocated all the memory the object needs, it only initialized the instance data local to rootclass. Now it's the icclass dispatcher's turn to do some work. It has to initialize the instance data that is local to icclass.

A dispatcher finds its local instance data by using the INST_DATA() macro (defined in <intuition/classes.h>):

```
void *INST_DATA(Class *localclass, Object *object);
```

INST_DATA() takes two arguments, a pointer to a class and a pointer to the object. The INST_DATA() macro returns a pointer to the instance data local to localclass. When the icclass dispatcher was called, it received three arguments, one of which was a pointer to the local class (icclass). The icclass dispatcher passes this pointer and the new object pointer it got from DoSuperMethodA() to INST_DATA() to get a pointer to the instance data local to icclass.

After initializing its local instance data, the icclass dispatcher passes control back to the modelclass dispatcher, which in turn, initializes the instance data local to modelclass. Finally, the rkmmodelclass dispatcher regains control and now has to take care of its local instance data.

To find its local instance data, the rkmmodelclass dispatcher needs a pointer to its Class and a pointer to the new object. The dispatcher function gets its Class pointer as its first argument (cl from the dispatchRKMModel() prototype above). It gets the new object pointer as the return value from DoSuperMethodA(). In this case, INST_DATA() returns a pointer to an RKMModData structure.

Now the dispatcher has to initialize its local instance data. It has to scan through the tag list passed in the OM_NEW message looking for initial values for the RKMMOD_CurrVal and RKMMOD_Limit attributes. As an alternative, the dispatcher's OM_NEW method can use its OM_SET method to handle initializing these "settable" attributes.

Finally, the dispatcher can return. When the dispatcher returns from an OM_NEW method, it returns a pointer to the new object.

If the OM_NEW method fails, it should tell the partially initialized object it got from its superclass's dispatcher to dispose of itself (using OM_DISPOSE) and return NULL.

## 1.21 12 / / Writing The Dispatcher / OM_SET/OM_UPDATE

For the OM_SET message, the rkmmodelclass dispatcher steps through the attribute/value pairs passed to it in the OM_SET message looking for the local attributes (see OM_NEW for the OM_SET message structure). The RKMMOD_Limit attribute is easy to process. Just find it and record the value in the local RKMModData.vallimit field.

Because the function of the rkmmodelclass's OM_SET and OM_UPDATE methods are almost identical, the rkmmodelclass dispatcher handles them as the same case. The only difference is that, because the OM_UPDATE message comes from another Boopsi object, the OM_UPDATE method can report on transitory state changes of an attribute. For example, when the user slides a Boopsi prop gadget, that prop gadget sends out an interim OM_UPDATE message for every interim value of PGA_Top. When the user lets go of the prop gadget, the gadget sends out a final OM_UPDATE message. The OM_UPDATE message is almost identical to the OM_SET message:

```
#define OPUF_INTERIM    (1<<0)

/* the OM_NOTIFY method uses the same structure */

struct opUpdate {
    ULONG             MethodID;
    struct TagItem    *opu_AttrList;
    struct GadgetInfo *opu_GInfo;
    ULONG             opu_Flags;      /* The extra field */
};
```

A dispatcher can tell the difference between an interim and final OM_UPDATE message because the OM_UPDATE message has an extra field on it for flags. If the low order bit (the OPUF_INTERIM bit) is set, this is an interim OM_UPDATE message. The interim flag is useful to a class that wants to ignore any transitory messages, processing only final attribute values. Because rkmmodelclass wants to process all changes to its attributes, it processes all OM_UPDATE messages.

The RKMMOD_CurrVal attribute is a little more complicated to process. The dispatcher has to make sure the new current value is within the limits set by RKMMOD_Limit, then record that new value in the local RKMModData.currval field. Because other objects need to hear about changes to RKMMOD_CurrVal, the dispatcher has to send a notification request. It does this by sending itself an OM_NOTIFY message. The OM_NOTIFY message tells an object to notify its targets (its ICA_TARGET and the objects in its broadcast list) about an attribute change. The OM_NOTIFY method does this by sending OM_UPDATE messages to all of an object's targets.

The rkmmodelclass dispatcher does not handle the OM_NOTIFY message itself. It inherits this method from modelclass, so the rkmmodelclass dispatcher passes OM_NOTIFY messages on to its superclass. To notify its targets, the rkmmodelclass dispatcher has to construct an OM_NOTIFY message. The OM_NOTIFY method uses the same message structure as OM_UPDATE. Using the stack-based version of DoSuperMethodA(), DoSuperMethod(), the dispatcher can build an OM_NOTIFY message on the stack:

```
     . . .

    struct TagItem tt[2];
    struct opUpdate *msg;
     . . .

    tt[0].ti_Tag  = RKMMOD_CurrVal;  /* make a tag list.  */
    tt[0].ti_Data = mmd->currval;
    tt[1].ti_Tag  = TAG_END;

    DoSuperMethod(cl, o, OM_NOTIFY, tt, msg->opu__GInfo,
                  ((msg->MethodID == OM_UPDATE) ? (msg->opu_Flags) : 0L));
     . . .
```

Because the OM_NOTIFY needs a tag list of attributes about which to issue updates, the dispatcher builds a tag list containing just the RKMMOD_CurrVal tag and its new value. The dispatcher doesn't use the tag list passed to it in the OM_UPDATE/OM_NOTIFY message because that list can contain many other attributes besides RKMMOD_CurrVal.

The msg variable in the DoSuperMethod() call above is the OM_SET or OM_UPDATE message that was passed to the dispatcher. The dispatcher uses that structure to find a pointer to the GadgetInfo structure that the OM_NOTIFY message requires. The GadgetInfo structure comes from Intuition and contains information that Boopsi gadgets need to render themselves. For the moment, don't worry about what the GadgetInfo structure actually does, just pass it on. The targets of an rkmmodel will probably need it.

Notice that the dispatcher has to test to see if the message is an OM_SET or OM_UPDATE so it can account for the opu_Flags field at the end of the OM_UPDATE message.

Processing the RKMMOD_Up and RKMMOD_Down attributes is similar to the RKMMOD_CurrVal attribute. When the dispatcher sees one of these, it has to increment or decrement the local RKMModData.currval, making sure RKMModData.currval is within limits. The dispatcher then sends an OM_NOTIFY message to the superclass about the change to RKMModData.currval.

The return value from the dispatcher's OM_SET method depends on the what effect the attribute change has to the visual state of the objects in the rkmmodel's broadcast list. If an attribute change will not affect the visual state of the rkmmodel's objects, the OM_SET method returns zero. If the attribute change could trigger a change to the rkmmodel's objects, it returns something besides zero. For example, the rkmmodelclass OM_SET method returns 1L if an rkmmodel's RKMMOD_CurrVal, RKMMOD_Up, or RKMMOD_Down attribute is changed.

At some point the rkmmodelclass dispatcher has to allow its superclasses to process these attributes it inherits. Normally a dispatcher lets the superclass process its attributes before attempting to process any local attributes. The rkmmodelclass dispatcher does this by passing on the OM_SET or OM_UPDATE message using DoSuperMethodA() (inheritance at work!). As an alternative, the dispatcher can use the amiga.lib function SetSuperAttrs(). See the amiga.lib Autodocs for more details on this function.

## 1.22 12 / / Writing The Dispatcher / OM_GET

```
The rkmmodel only has one "gettable" attribute: RKMMOD_CurrVal, which
makes processing it easy.  The OM_GET message looks like this (defined in
<intuition/classusr.h>):

    struct opGet {
        ULONG MethodID;     /* OM_GET */
        ULONG opg_AttrID;   /* The attribute to retrieve */
        ULONG *opg_Storage; /* a place to put the attribute's value */
    };

When the rkmmodelclass dispatcher receives an OM_GET message with an
opg_AttrID equal to RKMMOD_CurrVal, it copies the current value
(RKMModData).currval to the memory location opg_Storage points to and
returns a value of TRUE.  The TRUE indicates that there was no error.  If
opg_AttrID is not RKMMOD_CurrVal, the dispatcher should let its superclass
handle this message.

The rkmmodelclass dispatcher can take advantage of the fact that the only
"gettable" attribute available to an rkmmodel is RKMMOD_CurrVal (the
attributes defined by modelclass and icclass are not gettable--see the
Boopsi Class Reference in the Appendix B of this manual for more details
on which attributes are "settable", "gettable", etc.).  If opg_AttrID is
not RKMMOD_CurrVal, the rkmmodelclass dispatcher can return FALSE,
indicating that the attribute was not "gettable".

If the rkmmodelclass dispatcher comes across any other messages besides
OM_NEW, OM_SET, OM_UPDATE, and OM_GET message, it blindly passes them on
to its superclass for processing.
```

## 1.23 12 / / Writing The Dispatcher / Making the New Class

```
The Intuition function MakeClass() creates a new Boopsi class:

    Class *MakeClass(UBYTE *newclassID, UBYTE *pubsuperclassID,
                     Class *privsuperclass, UWORD instancesize,
                     ULONG flags);

If the new class is going to be public, newclassID is a string naming the
new class.  If the new class is private, this field is NULL.  The next two
fields tell MakeClass() where to find the new class's superclass.  If the
superclass is public, pubsuperclassID points to a string naming that
public superclass and the privsuperclass pointer is NULL.  If the
superclass is private, privsuperclass points to that superclass's Class
structure and pubsuperclassID is NULL. The size of the new class's local
instance data is instancesize.  The last parameter, flags, is for future
enhancement.  For now, make this zero.

If it is successful, MakeClass() returns a pointer to the new class,
otherwise it returns NULL.  When MakeClass() is successful, it also takes
measures to make sure no one can "close" the new class's superclass (using
FreeClass()).  It does this by incrementing a private field of the
superclass that keeps track of how many subclasses the superclass
```

currently has.

After successfully creating a class, an application has to tell the class where its dispatcher is. The Class pointer (defined in <intuition/classes.h>) returned by MakeClass() contains a Hook structure called cl_Dispatcher, which is used to call the dispatcher. The application has to initialize this hook:

```
myclass->cl_Dispatcher.h_Entry = HookEntry;
/* HookEntry() is defined in amiga.lib */

myclass->cl_Dispatcher.h_SubEntry = dispatchRKMModel;
```

The h_Entry field points to a function in amiga.lib that copies the function arguments to where the dispatcher expects them. See the "Callback Hooks" section of the "Utility Library" chapter of this manual for more details.

To make a class public instead of private, an application has to call AddClass() in addition to giving the class a name in MakeClass(). AddClass() takes one argument, a pointer to a valid Class structure that has been initialized as a public class by MakeClass(). To remove a public class added to the system with AddClass(), pass the public class pointer to RemoveClass(). See the Intuition Autodocs for more details on AddClass() and RemoveClass().

## 1.24 12 / / Writing The Dispatcher / RKMModel.c

The following code, RKMModel.c, makes up an initialization function and the dispatcher function for a private class informally called rkmmodelclass.

RKMModel.c

Below is a diagram showing how an application could use an rkmmodelclass object:

Figure 12-5: Rkmmodelclass Object Diagram

In this diagram, the application uses buttongclass Boopsi gadgets to send the rkmmodelclass the RKMMOD_Up and RKMMOD_Down attribute pulses.

The example takes advantage of an odd feature of buttongclass. When the user clicks on a buttongclass gadget, it sends an OM_UPDATE to its ICA_TARGET, even though no Boopsi attribute of buttongclass has changed. It does this because it’s a convenient way to report button clicks.

Whenever a gadget sends a notification, the list of attribute/value pairs in the OM_NOTIFY message always contains the gadget’s GA_ID. This is an easy way for the button to inform its target of its ID so the target knows which gadget sent the OM_UPDATE message. When a buttongclass sends a notification because of a button click, it only sends out an OM_UPDATE about its GA_ID because none of its attributes changed.

When the user clicks one of the buttons in the rkmmodelclass diagram, the

button uses an ICA_MAP to map its GA_ID to one of the "dummy" pulse attributes, RKMMOD_Up and RKMMOD_Down. When the rkmmodel receives the OM_UPDATE message about RKMMOD_Up or RKMMOD_Down, it increments or decrements its internal value.

There is one more important thing to note about rkmmodelclass. Looking at the rkmmodelclass Object diagram above, an rkmmodel's RKMMOD_CurrVal changes because it received an OM_UPDATE message from one of its gadgets. RKMMOD_CurrVal can also change if the application explicitly set RKMMOD_CurrVal using SetAttrs() or SetGadgetAttrs().

The primary difference between the OM_SET message that SetAttrs() sends and the OM_SET message that SetGadgetAttrs() sends is that SetAttrs() passes a NULL in opSet.ops_GInfo instead of a GadgetInfo pointer. This doesn't present a problem for the rkmmodel object, because it doesn't use the GadgetInfo structure. The problem is that when the rkmmodel notifies its targets, some of which are gadgets, they can't update their visual state because they need a GadgetInfo to render themselves. For this reason, the rkmmodelclass dispatcher returns a positive non-zero value when an attribute change occurs that could cause a change in the visual state of any objects in its broadcast list. An application that uses rkmmodelclass must test the return value when calling SetAttrs() on an rkmmodelclass object to tell if the attribute change requires a visual refresh of the gadgets (see the Intuition Autodocs for RefreshGadgets()).

> Boopsi Dispatchers Can Execute on Intuition's Context.
> ---
> Notice that the gadgets in the figure above send OM_UPDATE messages to the rkmmodel when the user manipulates them. Because Intuition handles the user input that triggers the OM_UPDATE messages, Intuition itself is sending the OM_UPDATE messages. This means the rkmmodelclass dispatcher must be able to run on Intuition's context, which puts some limitations on what the dispatcher is permitted to do: it can't use dos.library, it can't wait on application signals or message ports, and it can't call any Intuition functions which might wait on Intuition.

Although rkmmodelclass serves as an example of a class, it leaves a little to be desired in a real-world implementation. To create the "prop-integer-up/down" super gadget from the diagram above, the application has to create, initialize, and link nine Boopsi objects, which is tedious, especially if the application needs several of these super gadgets. Ideally, all these functions would be rolled into some subclass of gadgetclass. If there were such a class, an application would only have to create one instance of this subclass to get such a gadget. When the subclass received an OM_NEW message, it would take care of creating, initializing, and linking all of the Boopsi objects that make up the whole super gadget.

## 1.25 12 / Boopsi Class / White Boxes - The Transparent Base Classes

Boopsi gadgets and images were designed to be backwards compatible with the old Intuition Gadgets and Images, so as part of their instance data, both types of objects have the old Intuition structures built into them. When NewObject() creates a new gadget or image object, the pointer it

returns points to the object's embedded Gadget or Image corresponding structure. Because Intuition can tell the difference between Boopsi images and gadgets and the original images and gadgets, applications can use Boopsi images and gadgets interchangeably with the older Intuition entities.

Although normally considered a "programming sin", in some cases it is legal for class dispatchers to directly manipulate some internal fields of certain Boopsi objects. For compatibility reasons, a Boopsi image or gadget object contains an actual Image or Gadget structure. These objects are instances of the Transparent Base Classes, imageclass and gadgetclass.

To change an attribute of a Boopsi object, you normally invoke the set method, OM_SET. The Intuition functions SetAttrs() and SetGadgetAttrs() invoke this method. A Boopsi class is informed of any attribute change at that time, allowing it to react to this change. The reaction can include validating the changed attribute, changing other attributes to match, or informing other objects of the change. That is the inherent advantage of using function calls to change attributes.

When using conventional images and gadgets, you generally modify the structure's fields directly. This operation is very fast. For conventional images and gadgets, there is no class that needs to know about the changes, so there is no problem. However, this is untrue of Boopsi images and gadgets. Although directly modifying the Boopsi object's internal structure would provide a performance increase over using the Boopsi OM_SET mechanism, altering a Boopsi object's internal structure directly will not give the class the opportunity to react to any structure changes. This violates the Boopsi concept, and therefore cannot be done in general.

In order to provide a balance between the flexibility of function-access and the performance of direct-access, the transparent base classes imageclass and gadgetclass do not depend on being informed of changes to certain fields in the internal Image and Gadget structures. This means that it is OK for the dispatchers of direct subclasses of imageclass and gadgetclass to modify specific fields of Boopsi images or gadgets. Applications and indirect subclass dispatchers of imageclass or gadgetclass may not modify those fields, since their parent classes may depend on hearing about changes to these fields, which the SetAttrs() call (or a similar function) provides.

For dispatchers of direct subclasses of imageclass, the following are the only fields of the Image structure that are alterable:

```
LeftEdge      Width     ImageData
TopEdge       Height    PlanePick
PlaneOnOff
```

For dispatchers of direct subclasses of gadgetclass, the following are the only fields of the Gadget structure that are alterable:

```
LeftEdge    Flags           GadgetText
TopEdge     GadgetType      SpecialInfo
Width       GadgetRender    Activation
Height      SelectRender
```

Under no circumstances may an application or an indirect subclass modify one of these fields, even if the subclass knows the superclasses do not depend on notification for this field. This is the only way to preserve the possibility for future enhancements to that superclass. Note that these fields are not alterable while the gadget or image object is in use (for example, when it is attached to a window).

## 1.26 12 Boopsi--Object Oriented Intuition / Boopsi Gadgets

One of the major enhancements to Intuition for Release 2 is the implementation of customizable Boopsi gadgets. Boopsi gadgets are not limited by dependencies upon Intuition Image and Gadget structures. Unlike Release 1.3 gadgets, which were handled exclusively by Intuition, Boopsi gadgets handle their own rendering and their own user input.

Since Boopsi gadgets draw themselves, there is almost no restriction on what they can look like. A Boopsi gadget can use graphics.library RastPort drawing functions to draw vector-based imagery which the gadget can scale to any dimension. Instead of just a two-state Boolean gadget, a Boopsi gadget can have any number of states, each of which has its own imagery. If a programmer wanted to he could even make a Boopsi gadget that uses the animation system to render itself.

Because Boopsi gadgets handle their own input, they see all the user's input, which the gadget is free to interpret. While the user has a Boopsi gadget selected, the gadget can track mouse moves, process mouse and keyboard key presses, or watch the timer events.

The power of a Boopsi gadget is not limited to its ability to handle its own rendering and user input. Boopsi gadgets are also Boopsi objects so the gain all the benefits Boopsi provides. This means all Boopsi gadgets inherit the methods and attributes from their superclasses. Boopsi gadgets can use Boopsi images to take care of rendering their imagery. A Boopsi gadget could be a "composite" gadget that is composed of several Boopsi gadgets, images, and models.

The Boopsi Gadget Methods The Active Gadget

## 1.27 12 / Boopsi Gadgets / The Boopsi Gadget Methods

Intuition drives a Boopsi gadget by sending it Boopsi messages. Intuition uses a series of five Boopsi methods:

| | |
|---|---|
| GM_RENDER | This method tells the gadget to render itself. |
| GM_HITTEST | This method asks a gadget whether it has been "hit" by a mouse click. |
| GM_GOACTIVE | This method asks a gadget if it wants to be the active gadget. |
| GM_HANDLEINPUT | This method passes a gadget an input event. |

GM_GOINACTIVE  This method tells a gadget that it is no longer active.

The formats of each of these Boopsi messages differ, but they all have two things in common.  Like all Boopsi messages, each starts with their respective method ID.  For each of these methods, the method ID field is followed by a pointer to a GadgetInfo structure (defined in <intuition/cghooks.h>).  The GadgetInfo structure contains information about the display on which the gadget needs to render itself:

```
struct GadgetInfo {
    struct Screen          *gi_Screen;
    struct Window          *gi_Window;     /* null for screen gadgets */
    struct Requester       *gi_Requester;  /* null if not GTYP_REQGADGET */

    /* rendering information: don't use these without cloning/locking.
     * Official way is to call ObtainGIRPort()
     */
    struct RastPort        *gi_RastPort;
    struct Layer           *gi_Layer;

    /* copy of dimensions of screen/window/g00/req(/group)
     * that gadget resides in.  Left/Top of this box is
     * offset from window mouse coordinates to gadget coordinates
     *  screen gadgets:                 0,0 (from screen coords)
     *  window gadgets (no g00):        0,0
     *  GTYP_GZZGADGETs (borderlayer):  0,0
     *  GZZ innerlayer gadget:          borderleft, bordertop
     *  Requester gadgets:              reqleft, reqtop
     */
    struct IBox            gi_Domain;

    /* these are the pens for the window or screen */
    struct {
        UBYTE   DetailPen;
        UBYTE   BlockPen;
    }                      gi_Pens;

    /* the Detail and Block pens in gi_DrInfo->dri_Pens[] are
     * for the screen.  Use the above for window-sensitive colors.
     */
    struct DrawInfo        *gi_DrInfo;

    /* reserved space: this structure is extensible
     * anyway, but using these saves some recompilation
     */
    ULONG                  gi_Reserved[6];
};
```

All the fields in this structure are read only.

Although this structure contains a pointer to the gadget's RastPort structure, applications should not use it for rendering.  Instead, use the intuition.library function ObtainGIRPort() to obtain a copy of the GadgetInfo's RastPort.  When the gadget is finished with this RastPort, it

should call ReleaseGIRPort() to relinquish the RastPort.

```
 GM_RENDER     GM_GOACTIVE/GM_HANDLEINPUT
 GM_HITTEST    GM_GOINACTIVE
```

## 1.28 12 / / The Boopsi Gadget Methods / GM_RENDER

Every time Intuition feels it is necessary to redraw a Boopsi gadget, it sends a gadget a GM_RENDER message. The GM_RENDER message (defined in <intuition/gadgetclass.h>) tells a gadget to render itself:

```
    struct gpRender
    {
        ULONG             MethodID;   /* GM_RENDER */
        struct GadgetInfo *gpr_GInfo;
        struct RastPort   *gpr_RPort; /* all ready for use */
        LONG              gpr_Redraw; /* might be a "highlight pass" */
    };
```

Some events that cause Intuition to send a GM_RENDER are: an application passed the gadget to OpenWindow(), the user moved or resized a gadget's window, or an application explicitly asked Intuition to refresh some gadgets.

The GM_RENDER message contains a pointer to the gadget's RastPort so the GM_RENDER method does not have to extract it from the gpr_GInfo GadgetInfo structure using ObtainGIRPort()). The gadget renders itself according to how much imagery it needs to replace. The gpr_Redraw field contains one of three values:

GREDRAW_REDRAW  Redraw the entire gadget.

GREDRAW_UPDATE  The user has manipulated the gadget, causing a change to its imagery. Update only that part of the gadget's imagery that is effected by the user manipulating the gadget (for example, the knob and scrolling field of the prop gadget).

GREDRAW_TOGGLE  If this gadget supports it, toggle to or from the highlighting imagery.

Intuition is not the only entity that calls this method. The gadget's other methods may call this method to render the gadget when it goes through state changes. For example, as a prop gadget is following the mouse from the gadget's GM_HANDLEINPUT method, the gadget could send itself GM_RENDER messages, telling itself to update its imagery according to where the mouse has moved.

## 1.29 12 / / The Boopsi Gadget Methods / GM_HITTEST

When Intuition gets a left mouse button click in a window, one of the things it does is check through the window's list of gadgets to see if

that click was inside the bounds of a gadget's Gadget structure (using the LeftEdge, TopEdge, Width, and Height fields). If it was (and that gadget is a Boopsi gadget), Intuition sends that gadget a GM_HITTEST message (defined in <intuition/gadgetclass.h>):

```
    struct gpHitTest
    {
        ULONG             MethodID;     /* GM_HITTEST   */
        struct GadgetInfo *gpht_GInfo;
        struct
        {
            WORD X;     /* Is this point inside of the gadget? */
            WORD Y;
        } gpht_Mouse;
    };
```

This message contains the coordinates of the mouse click. These coordinates are relative to the upper-left of the gadget (LeftEdge, TopEdge).

Because Intuition can only tell if the user clicked inside gadget's "bounding box", Intuition only knows that the click was close to the gadget. Intuition uses the GM_HITTEST to ask the gadget if the click was really inside the gadget. The gadget returns GMR_GADGETHIT (defined in <intuition/gadgetclass>) to tell Intuition that the user hit it, otherwise it returns zero. This method allows a gadget to be any shape or pattern, rather than just rectangular.

## 1.30 12 / / The Boopsi Gadget Methods / GM_GOACTIVE/GM_HANDLEINPUT

If a gadget returns GMR_GADGETHIT, Intuition will send it a GM_GOACTIVE message (defined in <intuition/gadgetclass.h>):

```
    struct gpInput /* Used by GM_GOACTIVE and GM_HANDLEINPUT */
    {
        ULONG             MethodID;
        struct GadgetInfo *gpi_GInfo;
        struct InputEvent *gpi_IEvent;
                             /* The input event that triggered this method
                              * (for GM_GOACTIVE, this can be NULL)      */
        LONG              *gpi_Termination;
                             /* For GADGETUP IntuiMessage.Code           */
        struct
        {
            WORD X;          /* Mouse position relative to upper         */
            WORD Y;          /* left corner of gadget (LeftEdge,TopEdge) */
        } gpi_Mouse;
    };
```

The GM_GOACTIVE message gives a gadget the opportunity to become the active gadget. The active gadget is the gadget that is currently receiving user input. Under normal conditions, only one gadget can be the active gadget (it is possible to have more than one active gadget using a groupgclass object--See the Boopsi Class Reference in the Appendix B of this manual for more details).

While a gadget is active, Intuition sends it GM_HANDLEINPUT messages. Each GM_HANDLEINPUT message corresponds to a single InputEvent structure. These InputEvents can be keyboard presses, timer events, mouse moves, or mouse button presses. The message's gpi_IEvent field points to this InputEvent structure. It's up to the GM_HANDLEINPUT method to interpret the meaning of these events and update the visual state of the gadget as the user manipulates the gadget. For example, the GM_HANDLEINPUT method of a prop gadget has to track mouse events to see where the user has moved the prop gadget's knob and update the gadget's imagery to reflect the new position of the knob.

For the GM_GOACTIVE method, the gpi_IEvent field points to the struct InputEvent that triggered the GM_GOACTIVE message. Unlike the GM_HANDLEINPUT message, GM_GOACTIVE's gpi_IEvent can be NULL. If the GM_GOACTIVE message was triggered by a function like intuition.library's ActivateGadget() and not by a real InputEvent (like the user clicking the gadget), the gpi_IEvent field will be NULL.

For gadgets that only want to become active as a direct result of a mouse click, this difference is important. For example, the prop gadget becomes active only when the user clicks on its knob. Because the only way the user can control the prop gadget is via the mouse, it does not make sense for anything but the mouse to activate the gadget. On the other hand, a string gadget doesn't care how it is activated because, as soon as it's active, it gets user input from the keyboard rather than the mouse. Not all gadgets can become active. Some gadgets cannot become active because they have been temporarily disabled (their Gadget.Flags GFLG_DISABLED bit is set). Other gadgets will not become active because they don't need to process input. For example, a toggle gadget won't become active because it only needs to process one input event, the mouse click that toggles the gadget (which it gets from the GM_GOACTIVE message). If a toggle gadget gets a GM_GOACTIVE message and its gpi_IEvent field is not NULL, it will toggle its state and refuse to "go active".

The GM_GOACTIVE method has to take care of any visual state changes to a gadget that a GM_GOACTIVE message might trigger. For example, the toggle gadget in the previous paragraph has to take care of toggling its visual state from selected imagery to unselected imagery. If the gadget goes through a state change when it becomes the active gadget, (like when a string gadget positions its cursor) GM_GOACTIVE has to take care of this.

The return values of both GM_GOACTIVE and GM_HANDLEINPUT tell Intuition whether or not the gadget wants to be active. A gadget's GM_GOACTIVE method returns GMR_MEACTIVE (defined in <intuition/gadgetclass.h>) if it wants to become the active gadget. A gadget's GM_HANDLEINPUT method returns GMR_MEACTIVE if it wants to remain the active gadget. If a gadget either does not want to become or remain the active gadget, it returns one of the "go inactive" return values:

GMR_NOREUSE  Tells Intuition to throw away the gpInput.gpi_IEvent InputEvent.

GMR_REUSE  Tells Intuition to process the gpInput.gpi_IEvent InputEvent.

GMR_NEXTACTIVE  Tells Intuition to throw away the gpInput.gpi_IEvent

```
                InputEvent and activate the next GFLG_TABCYCLE gadget.

GMR_PREVACTIVE  Tells Intuition to throw away the gpInput.gpi_IEvent
                InputEvent and activate the previous GFLG_TABCYCLE
                gadget.
```

GMR_NOREUSE tells Intuition that the gadget does not want to be active and to throw away the InputEvent that triggered the message. For example, an active prop gadget returns GMR_NOREUSE when the user lets go of the left mouse button (thus letting go of the prop gadget's knob).

For the GM_HANDLEINPUT method, a gadget can also return GMR_REUSE, which tells Intuition to reuse the InputEvent. For example, if the user clicks outside the active string gadget, that string gadget returns GMR_REUSE. Intuition can now process that mouse click, which can be over another gadget. Another case where a string gadget returns GMR_REUSE is when the user pushes the right mouse button (the menu button). The string gadget becomes inactive and the menu button InputEvent gets reused. Intuition sees this event and tries to pop up the menu bar.

For the GM_GOACTIVE method, a gadget must not return GMR_REUSE. If a gadget gets a GM_GOACTIVE message from Intuition and the message has an gpi_IEvent, the message was triggered by the user clicking on the gadget. In this case, Intuition knows that the user is trying to select the gadget. Intuition doesn't know if the gadget can be activated, but if it can be activated, the event that triggered the activation has just taken place. If the gadget cannot become active for any reason, it must not let Intuition reuse that InputEvent as the gadget has already taken care of the the event's purpose (clicking on the gadget). In essence, the user tried to activate the gadget and the gadget refused to become active.

The other two possible return values, GMR_NEXTACTIVE and GMR_PREVACTIVE were added to the OS for Release 2.04. These tell Intuition that a gadget does not want to be active and that the InputEvent should be discarded. Intuition then looks for the next (GMR_NEXTACTIVE) or previous (GMR_PREVACTIVE) gadget that has its GFLG_TABCYCLE flag set in its Gadget.Activation field (see the gadgetclass GA_TabCycle attribute in the Boopsi Class Reference in the Appendix B of this manual).

For both GM_GOACTIVE and GM_HANDLEINPUT, the gadget can bitwise-OR any of these "go inactive" return values with GMR_VERIFY. The GMR_VERIFY flag tells Intuition to send a GADGETUP IntuiMessage to the gadget's window. If the gadget uses GMR_VERIFY, it has to supply a value for the IntuiMessage.Code field. It does this by passing a value in the gpInput.gpi_Termination field. This field points to a long word, the lower 16-bits of which Intuition copies into the Code field. The upper 16-bits are for future enhancements, so clear these bits.

## 1.31 12 / / The Boopsi Gadget Methods / GM_GOINACTIVE

After an active gadget deactivates, Intuition sends it a GM_GOINACTIVE message (defined in <intuition/gadgetclass.h>):

```
    struct gpGoInactive
    {
```

```
        ULONG             MethodID;    /* GM_GOINACTIVE */
        struct GadgetInfo *gpgi_GInfo;

        /* V37 field only!  DO NOT attempt to read under V36!          */
        ULONG             gpgi_Abort; /* gpgi_Abort=1 if gadget was    */
                                      /* aborted by Intuition          */
                                      /* and 0 if gadget went inactive */
                                      /* at its own request.           */
    };
```

The gpgi_Abort field contains either a 0 or 1. If 0, the gadget became inactive on its own power (because the GM_GOACTIVE or GM_HANDLEINPUT method returned something besides GMR_MEACTIVE). If gpgi_Abort is 1, Intuition aborted this active gadget. Some instances where Intuition aborts a gadget include: the user clicked in another window or screen, an application removed the active gadget with RemoveGList(), and an application called ActiveWindow() on a window other than the gadget's window.

## 1.32 12 / Boopsi Gadgets / The Active Gadget

While a gadget is active, Intuition sends it a GM_HANDLEINPUT message for every timer pulse, mouse move, mouse click, and key press that takes place. A timer event pulse arrives about every tenth of a second. Mouse move events can arrive at a much higher rate than the timer pulses. Without even considering the keyboard, a gadget can get a lot of GM_HANDLEINPUT messages in a short amount of time. Because the active gadget has to handle a large volume of GM_HANDLEINPUT messages, the overhead of this method should be kept to a minimum.

Because the gadget will always receive a GM_GOACTIVE message before it is active and a GM_GOINACTIVE message after it is no longer active, the gadget can use these methods to allocate, initialize, and deallocate temporary resources it needs for the GM_HANDLEINPUT method. This can significantly reduce the overhead of GM_HANDLEINPUT because it eliminates the need to allocate, initialize, and deallocate resources for every GM_HANDLEINPUT message.

Note that the RastPort from ObtainGIRPort() is not cachable using this method. If the GM_HANDLEINPUT method needs to use a RastPort, it has to obtain and release the RastPort for every GM_HANDLEINPUT message using ObtainGIRPort() and ReleaseGIRPort().

RKMButtonclass.c

## 1.33 12 / / The Active Gadget / RKMButtonclass.c

The following example is a sample Boopsi gadget, RKMButClass.c. While the user has the RKMButton selected, the gadget sends an OM_UPDATE message to its ICA_TARGET for every timer event the button sees. The gadget sends notification about its RKMBUT_Pulse attribute, which is the horizontal distance in screen pixels the mouse is from the center of the button. The

gadget takes care of rendering all of its imagery (as opposed to using a Boopsi image to do it). The gadget's imagery is scalable to any dimensions and can be set (using SetGadgetAttrs()) while the gadget is in place.

One possible use for such a gadget is as buttons for a prop gadget. If the user has the prop gadget's RKMButton selected, while the mouse is to the left of the button's center, the knob on the prop gadget moves left. While the mouse is to the right of the button's center, the knob on the prop gadget moves right. The speed at which the knob moves is proportional to the horizontal distance from the mouse to the active RKMButton.

RKMButClass.c

## 1.34 12 Boopsi--Object Oriented Intuition / Function Reference

The following are brief descriptions of the Intuition and amiga.lib functions discussed in this chapter. See the "Amiga ROM Kernel Reference Manual: Includes and Autodocs" for details on each function call. All these functions require Release 2 or a later version of the Amiga operating system.

Table 12-1: Intuition Library Boopsi Functions

| Function | Description |
|---|---|
| NewObjectA() | Create a new Boopsi object (tag array form). |
| NewObject() | Create a new Boopsi object (varargs form). |
| DisposeObject() | Dispose of a Boopsi object. |
| SetAttrs() | Set one or more of a Boopsi object's attributes (tag array form). |
| SetGadgetAttrs() | Set one or more of a Boopsi object's attributes (varargs form). |
| GetAttr() | Obtain an attribute from a Boopsi object. |
| MakeClass() | Create a new private or public Boopsi class. |
| FreeClass() | Free a Boopsi class created by MakeClass(). |
| AddClass() | Add a public Boopsi class to Intuition's internal list of public classes. |
| RemoveClass() | Remove a public Boopsi class that was added to Intuition's internal list with AddClass(). |
| ObtainGIRPort() | Set up a RastPort for use by a Boopsi gadget dispatcher. |
| ReleaseGIRPort() | Free a RastPort set up by ReleaseGIRPort(). |

Table 12-2: Amiga.lib Boopsi Functions

| Function | Description |
|---|---|

| | |
|---|---|
| DoMethodA() | Send a Boopsi message to a Boopsi object (tag array form). |
| DoMethod() | Send a Boopsi message to a Boopsi object (varargs form). |
| DoSuperMethodA() | Send a Boopsi message to a Boopsi object as if the object was an instance of its class's superclass (tag array form). |
| DoSuperMethod() | Send a Boopsi message to a Boopsi object as if the object was an instance of its class's superclass (varargs form). |
| CoerceMethodA() | Send a Boopsi message to a Boopsi object as if the object was an instance of the specified class (tag array form). |
| CoerceMethod() | Send a Boopsi message to a Boopsi object as if the object was an instance of the specified class (varargs form). |
| SetSuperAttrs() | Send a Boopsi OM_SET message to the Boopsi object's superclass. |