CMU-ITC-86-052

The Andrew System

Programmer's Guide to the Base Environment, Version 2

Draft Version. Subject to Revision.

Prepared by

Ayami Ogura and Chris Neuwirth

Information Technology Center (ITC)
Carnegie Mellon University
Pittsburgh, PA 15213

A Joint Computing Venture
between IBM and
Carnegie Mellon University

Table of Contents

Introduction

This manual describes the Andrew system's *Base Environment, Version 2*, or BE2, a collection of routines that forms the base for all user interface and application program development in the Information Technology Center's (ITC) Andrew system. Andrew is a collaborative project between IBM and Carnegie-Mellon University (CMU) to create a computing and communication system that meets the needs of universities.

BE2 allows programmers to create (1) stand-alone applications that integrate text, graphics and images in a standard, efficient user interface; and (2) multi-media editors, i.e., editors that allow users to edit text, equations, graphs, tables, pictures, etc., all in a single program.

BE2 supports two capabilities that have not been available in the same system before: (1) the creation and editing of different types of objects in one place, and (2) the inclusion of arbitrary types of objects within the same editor. Some systems allow users to create and edit multiple objects within the same editor, but the set of objects that the editor can manage is fixed in advance. Users who need objects that are specialized (e.g., musical symbols) are unlikely to find their needs met. Other systems allow users to include arbitrary objects within an editor, but the objects are reduced to picture format, a representation that removes all but the most basic editing capabilities. In these systems, users who need to work with multiple objects do so by creating and editing the object in a specialized object-editor, then including the object in the target editor. If users need to edit the included object at some later time, they typically delete the object from the target editor, return to the editor that specializes in the object to edit it, then re-include the object in the target editor. The primary advantage of BE2 is that it allows programmers to build applications that support both activities--editing different types of objects in one place and editing arbitrary types of objects.

This document is divided into the following six sections:

1. Theory
2. Basics
3. Programming Environment
4. Data Streams
5. Routines, Objects, and Support Packages
6. Up and Coming

Theory presents an overview of BE2. It elaborates the goals of BE2 and discusses its design.

Basics introduces the two basic building blocks in BE2: the inset object and the data object. Inset objects manage the display of information on the screen and interaction with users; data objects manage the storage of information and its manipulation. The discussion of inset and data objects centers around nine example programs. These programs are intended to introduce you to the basic control and data structures you need to begin working with BE2. The example programs and explanations cannot tell the entire story for every BE2 routine, however, Before extrapolating from the material in Basics, you should at least skim the Section 5, *Routines, Objects, and Support Packages.*

Programming Environment discusses the environment in which BE2 applications are compiled, linked and loaded. To support a system that allows arbitrary objects to be included within the same program, BE2 provides a dynamic linking and loading subsystem.

Data Streams discusses the data stream representation developed to support arbitrary objects.

Routines, Objects, and Support Packages describes the entire set of BE2 routines. It is composed of several subsections:

a. Inset Routines
b. Inset/Window Management Routines
c. Data Object Routines
d. Menu Routines
e. The KeyMap Package
f. Document Objects
g. Dialog Inset and Manager
h. Scroll Bar Inset
i. Layout Pairs
j. Buffer Package
k. Update List Package
l. Key Recording Package

Routines are part of BE2 proper. Objects, e.g., *Document Objects*, are BE2 applications of general utility. Packages are combinations of data structures, routines, and protocols intended to simplify the creation of BE2 applications.

Up and Coming describes some of the routines and packages in the works.

The BE2 developers are currently specifying a set of graphics operations that are intended to shield application programmers from the underlying window management system. Mediating graphics calls through the BE2 graphics facility maximizes the device independence and portability of application programs. The graphics operations will be based on the operations provided by the X.11 window management system. *Appendix I: The BE2 -- X.11 Interface*, provides a detailed description of the conversion of BE2 graphics calls to X.11 calls. For more information on X.11, see Bettys, J., Newman, R. & Fera, T.D., *Xlib-C Language X Interface Protocol Version 11*.

This manual assumes that you have read the *User's Guides for Andrew* or have comparable knowledge of the Andrew system. It also assumes you know how to program in C, but it does not assume you are an experienced C programmer.

Because the Andrew system software, including its documentation, is still under development, you may encounter problems or require help. If you do, you should contact an Andrew user consultant.

Additional sources of information about BE2 can be found in *BE2: A Technical Overview*, located in */usr/andrew/doc/be2*, and in the ITC source code, located in */usr/andrew/src/be2*.

Credits

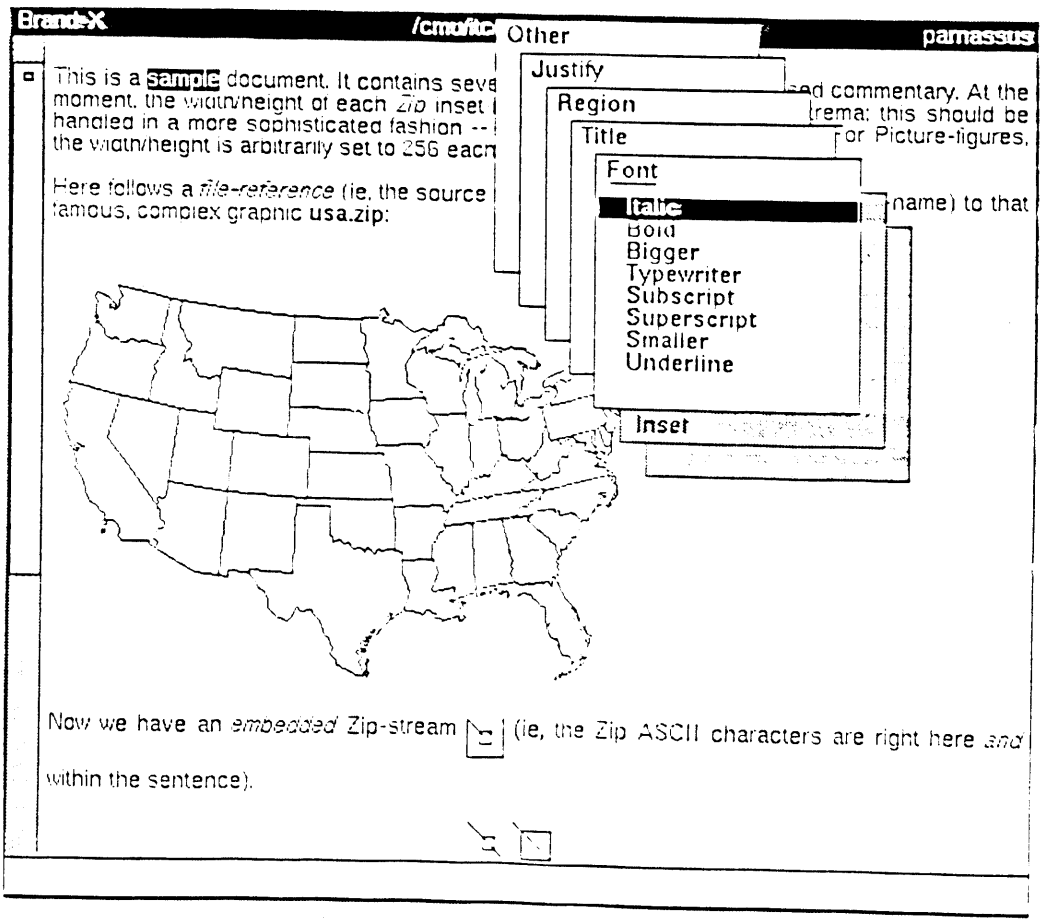
BE2 has been a collaborative effort of a large group of people at the Information Technology Center. The initial design team included: Andrew Palay, Wilfred J. Hansen, Mike Kazar, Bruce Lucas and Andrew Appel. Mike Kazar built the initial BE2 system from that design. The current design and implementation group includes: Mike Kazar, Andrew Palay, Mark Sherman, Maria Wadlow, Zalman Stern with assistance from Nathaniel Borenstein, Richard Cohn, Wilfred J. Hansen, David Nichols and Tom Neuendorfer.

BE2 would never have been developed without the prototype environment, BE1, developed at the ITC by James Gosling. That system was extended and greatly improved by Wilfred J. Hansen.

Theory

This section describes the goals for BE2 and the design decisions made to achieve those goals. BE2 has two primary goals: (1) to support the development of stand-alone applications that integrate text, graphics and images in a standard, efficient user interface; and (2) to support the development of multi-media editors, i.e., editors that allow users to edit text, equations, graphs, tables, pictures, etc., all in a single program.

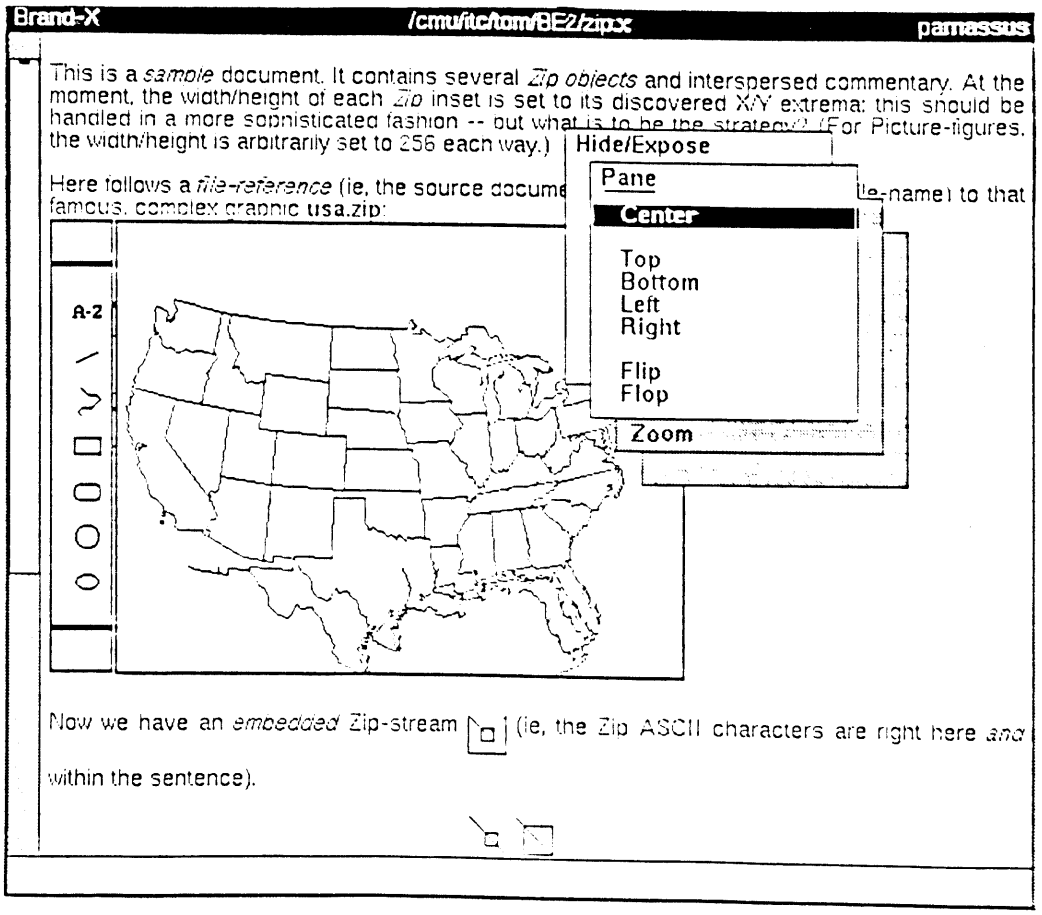
For example, the example below shows a document, based upon BE2, that contains text and a picture:



Sample BE2 Document with Text Editing Menus

The text and picture are separate objects and can be manipulated independently. This is easily seen by the different menus available. A mouse

hit in the text region, as shown above, will give you a full set of menu of options for editing the text. The next figure shows the same BE2 document with the picture inset being acted upon.



BE2 Document with Picture Editing Menus

Note that a mouse hit in the picture will give you a different set of menus for editing the picture.

In BE2, documents are not necessarily comprised of text alone. Documents may contain diagrams, tables, graphs, and pictures, as well as other graphical images. In this sense of the word *document*, a document may have no text at all, as in a "picture document" or a "table document." BE2 supports the creation of programs that allow users to create documents with a variety of images, both textual and graphical, in any combination. Text,

pictures, tables, graphs, equations, even the scroll bar are all kinds of *objects*. A document with text and a graph, therefore, has two objects, a text object and a graph object. A document that has only a picture and nothing else, has only one object -- the picture.

In addition to editing different type of objects in one place, BE2 supports the development of application programs that can include arbitrary objects upon demand. For example, *Brand-X*, a multi-media editor based upon BE2, can dynamically load any object that has been created according to BE2 protocols. The editor does not need to know about the object in advance (See *Example 2* in the section *Basics*, p. 25, for a demonstration).

Thus, BE2 supports two capabilities that have not been available in the same system before: (1) the creation and editing of different types of objects in one place, and (2) the inclusion of arbitrary types of objects within the same editor or application program. Some systems allow users to create and edit multiple objects within the same editor, but the set of objects that the editor can manage is fixed in advance. Users who need objects that are specialized (e.g., musical symbols) are unlikely to find their needs met. Other systems allow users to include arbitrary objects within an editor, but the objects are reduced to picture format, a representation that removes all but the most basic editing capabilities. In these systems, users who need to work with multiple objects do so by creating and editing the object in an specialized object-editor, then including the object in the target editor. If users need to edit the included object at some later time, they typically delete the object from the target editor, return to the editor that specializes in the object to edit it and then re-include the object in the target editor. The primary advantage of BE2 is that it allows programmers to build applications that support both activities--editing different types of objects in one place and editing arbitrary types of objects.

To provide these capabilities, BE2 is an open design. Application programmers can design and implement new objects that will eventually appear inside a multi-media editor or another application program. Application programmers must follow a set of guidelines when developing an object. If a programmer follows these guidelines, then the object can be included in any other application program or multi-media editor that has been developed to manage arbitrary objects. An object can be placed inside another object without either one having specific information about the other. The only piece of information that the enclosing object must have about the enclosed object is its name.

BE2 utilizes two major *classes* of objects. One object, the *data object*, contains the actual data to be displayed as well as routines that are used to manipulate that data. The other class of object, the *inset object*, or just *inset*, contains information on how to display that data on the screen, as well as routines that are used to manipulate that display. Both classes of objects have an associated set of routines that must be supplied with the objects. These routines provide a general mechanism for communicating among these objects. For example, the data object needs to provide routines to read and write a description of its contents to a file; the inset must provide

routines to update the display, handle key and mouse input, etc. These general routines will be discussed in detail in the sections *Basics* and *Routines, Objects, and Support Packages* of this manual.

Along with the set of general routines, an object may provide a set of specific routines. In the case of text, there is a data object that contains the characters that make up the text and routines that can, for example, insert a string, delete characters and return the character at a given position. There is also an associated inset object that describes how the text should be displayed on the screen. This includes information such as which document data object this inset is displaying, where the caret is currently located in the text what part of the text is currently visible on the screen. The inset may also provide routines that move the caret forward (or backward) one character and scroll forward (or backward) one screen. All the text data objects and all the text inset objects together form the package of routines that is known as the text object.

It is important to note that while there are only two *classes* of objects (insets and data objects), there are many *kinds* of objects (text object, graph object, etc.). Many of these, like the text object, involve both insets and data objects, and are therefore often referred to as a object *package*.

Object packages, the specific routines used for a certain kind of object, are not always necessary. Using only the general routines we can build a mechanism for embedding one object within another. Consider a document that contains both a line drawing and an equation. The interactions between the document objects and the line drawing objects are all done via the general routines. Other than the creation of the line drawing objects, the document objects do not care what kind of objects they are handling. Thus, you can write your own support packages to work with the general routines instead of relying on or waiting for such packages to be written by the Andrew system developers.

The next chapter in this manual will describe how insets and data objects are used.

A Note for Advanced Programmers

If you are planning on creating many of your own support packages, the following considerations that went in to the BE2 system design may be useful for you to know.

1) The design of the system stays away from the issue of the underlying window management system that will be used. The window management system is responsible for two major tasks: allocation of the screen and the lowest level graphic interface to the display. At this point we are unconcerned with how those two operations are handled. The BE2 graphics facility is intended to shield application programmers from window management details.

2) The design also stays away from trying to decide how the actual user interface should look. Of course, the document editor will

probably look very much like the current EditText, but that is not mandatory. In fact, by allowing the inclusion of one kind of object within another, a whole new set of decisions must be made: for example, the proper way to handle mouse clicks to a subordinate inset. These decisions can only be made after some reasonable experience with this system. Overall, a consistent user interface will most likely be provided by a set of user interface design guidelines and a very large tool kit.

3) It is necessary to look at the issue of higher level application programmer interfaces to this system (CMU-Tutor is one such example). While this is important, it is a separable issue. It is necessary to provide support for those developments, but the actual creation of a single application programmer interface is extremely doubtful. These interfaces must exist at different levels and will be created with different models in mind.

While each of these three topics is important to examine, these topics will not be covered in this document, since they will not affect the average user in the short run.

Basics

This section introduces the basics of BE2. It begins by introducing the two building blocks of BE2: the inset and the data object. Then, the basic routines of BE2 itself are introduced through several example programs. These examples illustrate the program structure needed by any application that uses BE2:

Example 1: Putting an inset in a window
Example 2: Dynamically loading an inset
Example 3: Responding to mouse hits
Example 4: Working with menus
Example 5: Responding to keyboard input
Example 6: Mapping keys to commands
Example 7: Working with a scroll bar

Example 1 introduces the routines, program structure and compilation techniques needed to put a simple inset in a window as a stand-alone application, i.e., an inset that will be running as an independent application program. Example 2 introduces the additional routines and program structure needed to dynamically load an inset. Example 3 introduces the routines and program structure needed to respond to mouse hits. Example 4 illustrates how to add menus to the inset. Example 5 describes how to respond to keyboard input from the user. Example 6 introduces facilities for mapping keys to commands. Example 7 describes how to work with a scroll bar.

After reading *Basics*, you will know the basic data and control structures needed to create a application program that displays multiple objects and allows the user to manipulate them. (Note: Examples showing how to work with data objects, documents, and other routines, obejcts, and packages, are forthcoming).

Insets and data objects

An inset is simply a rectangle of pixels in a window together with the associated code (called an *inset driver* or *driver*) that manages the display within that rectangle. The inset driver includes display routines, routines for initializing menus, routines for handling keyboard and mouse input, and so on. Examples of insets are a scroll of text, a picture embedded in a piece of text, a label in a drawing, a table in a piece of text, or an equation included in a document.

The inset is the first part of a general architecture for creating interactive applications in the Andrew system. The data object is the second part of this general architecture. Typically, an inset presents a view of a data object. In the above example, the data objects are streams of text with formatting information; equations; and tables. The data objects are displayed in the window either because a user wishes to view the information represented by the data objects or because he wishes to change (edit) the information. Whereas the inset is responsible for handling the display of information, the data object is responsible for maintaining that information, including storing

and manipulating it. For example, the data object associated with a document would contain the text of the document and procedures to manipulate that text. In general, you can think of data objects as a representation of permanent memory/storage, whereas insets represent views onto data objects.

The reason for having both insets and data objects is to provide a way to have multiple insets viewing a single data object. In this way, we can have, for example, a document that contains both a graph inset and a table inset that use the same underlying table data object. When the user updates the numbers in the table, the graph would adjust accordingly. Likewise, having both insets and data objects makes it possible to have a text editor that has multiple windows in which the user can edit the same document in more than one window. In this case, the data object would have to be the same, while two different insets view it. If insets and data objects were not separate, these application programs would not be possible.

The general interface that data objects provide is the minimal interface required for communication between two data objects and between an inset and a data object. Also, both insets and data objects can be nested.

The architecture is an open one, in that it specifies the minimal interface between the components of the system, such as text editors, drawing editors, and dialog boxes, to allow them to cooperate in creating the image the user sees and manipulates. Due to this open architecture, programmers are not confined to the available interfaces; they can choose or create their own. Similarly, they are not limited to the available objects.

This kind of mechanism is especially feasible for the creation of multi-media editors (a text editor that allows the inclusion of drawings in the text, a drawing editor that allows the inclusion of text in the drawing, etc.), and the creation of building blocks that can be used in educational applications, the advantage here being that the standard interface will make it easier for programmers to put them together and make interesting programs.

Example 1: Putting an inset in a window

This section describes how to write a program that uses a very simple inset. The program will create a new window and put the inset in the window; then the inset will draw *hello world* in the center of it. The program illustrates

-- declaring a structure for the inset,

-- defining driver routines for the inset, i.e., the set of routines for managing the inset's display,

-- exporting the driver routines and importing BE2 routines,

-- setting up an inset as a stand-alone application program, and

-- compiling the inset for static linking, i.e., linking before run-time.

After reading Example1, you will know the very basic program structure needed to work with insets in a stand-alone application program. Example 2 will introduce the changes you need to make in order to build an inset that can be dynamically loaded into a multi-media editor or other BE2 application.

The discussion that follows presents a step by step description of how to write the example program. If you were to follow the steps, you would produce a program, called *helloworld*, in four files:

-- a helloworld.h file -- will contain the data structure declarations for the inset, together with the export declarations for the driver routines.

-- a helloworld.c file -- will contain statements that import BE2 routines, export the driver routines, and define the driver routines.

-- main.c -- will contain declarations needed by BE2 linking and loading facilities, and statements that create an instance of the *helloworld* inset, create a window, put the inset in the window, and enter an interaction loop that mediates communication between a user and the application program.

-- Makefile -- will contain the directions for compiling, linking and loading the inset and main program.

For a complete listing of these files, see the subsection, *Program Listing for Example 1* (p. 23). On a first reading of this section, you may find it useful to just skim the program listing, then refer to the listing when needed as you study how to build the program. The source code is also available in the directory */usr/andrew/doc/be2/example1*, together with the compiled program.

Although the discussion of the steps relies heavily on the example, the steps apply generally to any inset that will be used in a stand-alone application.

Running the example program

Before reading the discussion of the example program, you may find it helpful to run the program on your workstation.

Action 1. To run the program, at the **Typescript** prompt, type

/usr/andrew/doc/be2/example1/helloworld

and press the Enter key.

Response. The program will produce a window with *hello world* centered in the body.

Action 2. Re-shape the program's window.

Response. The inset will respond to an update request and redraw *hello world* in the center.

Action 3. Click with the left mouse button.

Response. The program will make no response. This simple inset does not respond to keyboard or mouse input, and it has no menus. Later examples will illustrate how to extend the program so that the inset responds to user input.

Action 4. To quit the program, move the cursor to the window title bar, pop-up the menus and choose **Zap** from the card labeled *This Window*.

Response. The window will disappear from the screen.

Declaring a structure for an inset

Deciding on a name for the inset

To create an inset, you must decide on a name for it. The name is important for two reasons. First, if your inset is going to be included in a multi-media editor, the user will type your inset's name in order to obtain an instance of it. The name you pick should be unique and not conflict with already existing insets. (To see the list of existing insets, see the directory /usr/andrew/lib/be2.) The second reason the name of your inset is important is that the routines in the inset driver module will all be located in a *.c* file. For compiling your inset correctly, it is easiest if the module name for this *.c* file is the same as your inset name. For example, the routines for an inset named *view* would all be located in a file named *view.c*.

For Example 1, we chose the inset name, *helloworld*.

Declaring the data structure for the inset

The second thing you will want to do is to think about what data structure your inset needs and declare it in a *.h* file that has the same name as the inset you are building. In this example, the information is declared in a *helloworld* structure in the file *helloworld.h:*

```
struct helloworld {
    struct inset myinset; /* declare a BE2 inset */

};
```

The statement *struct inset myinset* declares *myinset* as a member of the *helloworld* structure of type *struct inset*, a structure provided by BE2.

In general, to declare an inset structure, you must declare a *struct inset*, optionally followed by any data that is specific to the inset you are creating:

```
struct <name of inset> {
      struct inset <my inset>; /* declare a BE2 inset */

      /* data specific to this inset */
};
```

struct inset is explained in detail in the section, *Inset Routines* (p. 91). Briefly, *struct inset* stores the following information:

(1) the name of the inset (e.g., helloworld),

(2) a pointer to the set of procedures for managing the inset (e.g., routines for creating an instance of the inset, redrawing the display, etc.), and

(3) the coordinates and attributes of the rectangle on the display that the inset manages.

You can also store data that is specific to the inset (e.g., a pointer to the data object associated with the inset; the position of the text carat for a text inset, etc.). Example 1 has no data that is specific to the inset. Example 3 will introduce you to working with data that is inset specific.

Defining the driver routines for the inset

The driver for an inset is the program module that manages the user display and interaction for instances of the inset. Each driver must define a set of driver routines. For an inset named *x* these should be defined in a module named *x.c* and must be named *x_New*, *x_FullUpdate*, *x_Update*, etc., where each routine name (e.g., *New*, *FullUpdate*, *Update*) is preceded by the inset name, followed by an underscore (e.g., *x_*). For the *helloworld* inset, these routines will be defined in *helloworld.c* and will be declared as *helloworld_New*, *helloworld_FullUpdate*, etc. (See the subsection *Program Listing for Example 1*, or /usr/andrew/doc/be2/example1).

The routines you define in the inset driver constitute entry points to the inset. These entry points will be expected by any program--either a stand-alone application or a multi-media editor--that uses the inset. It is important to note that you should never call the driver routines directly, even in the inset driver module. Instead, you should use a set of corresponding calling routines provided by BE2. BE2 routines are of the form *inset_* followed by the corresponding procedure name (e.g., *inset_New*, *inset_FullUpdate*). These BE2 routines, which take particular insets as arguments, provide an interface to an inset. Using BE2 routines makes it easier for a programmer to call insets as well as to program them, since the *inset_* routines guarantee that state information needed by the inset will be correct (See *Inset Routines*, p. 91).

Inset drivers generally will have more routines, or entry points, than just the required ones. Typically, you would create these if you need to allow users to manipulate the inset or its data object. For instance, a text editor needs to allow users to move the caret, and a drawing program may allow drawing of lines or movement of a point. But these entry points are *not* part of the standard inset interface; the standard inset interface contains only those routines necessary to allow the inclusion of an inset by programs that follow the protocols defined in BE2; these programs will have no specific knowledge about the inset.

The following discussion illustrates how to define the inset driver routines for the *helloworld* inset. You should note, however, that not all the possible inset driver routines are used in this example. Some inset driver routines have defaults which are used when no more specific routine is defined. For a more complete discussion of the inset driver routines and their defaults, see section *Inset Routines* (p. 91).

Example 1 requires three driver routines: a routine to create instances of the inset, a routine to initialize an instance of the inset, and a routine to respond to requests to do a full update of the inset's drawing. These routines constitute a minimal set that you would need to define if you are creating an inset.

Writing the inset creation routine

```
struct helloworld *helloworld_New()  {

    struct helloworld *hw;

    hw = (struct helloworld *) malloc(sizeof(struct helloworld));
    helloworld_Init(hw);
    return (hw);
}
```

This routine creates and initializes an instance of a *helloworld* inset. First, it dynamically allocates memory (see *malloc (3)* in the online help pages or Unix manual) and assigns the allocated space to a helloworld pointer, *hw*. Then it calls the initialization routine *helloworld_Init*. Finally, it returns a pointer to the inset that it has created.

Normally, the routine for creating an inset is passed a pointer to a data object, *dop*. Since this helloworld example does not have a data object, it is not used. (See Example 8, forthcoming, for a more typical creation routine).

You must provide an *x_New ()* routine for any inset *x* that you want to create. Basically, the routine should allocate storage for the inset and assign it to a pointer, call *x_Init* to initialize the inset, and return the pointer to the newly allocated and initialized inset.

Writing the routine for initializing the inset

```
helloworld_Init(hw)
struct helloworld *hw;  {

    inset_InitStructure("helloworld", hw);
}
```

helloworld_Init calls *inset_InitStructure* with the name of the newly created inset's type, *"helloworld"* and a pointer to the newly created inset, *hw*. *inset_InitStructure* records the name in *helloworld*'s inset structure and initializes other general inset information.

Normally, the routine for initializing an inset is passed a pointer to a data object, *dop*. Since this helloworld example does not have a data object, it is not used. (See Example 8, forthcoming, for a more typical initialization routine).

You must provide an *x_Init ()* routine for any inset *x* that you want to create. Before performing any other initializations, the routine must call *inset_InitStructure ("x," xp)* in order to initialize the newly created inset's general inset information. Then it should initialize any information that is specific to the inset *x*.

Writing a routine for full update requests

```
helloworld_FullUpdate(hw, how)
struct helloworld *hw;
int how;{

int x,y;

x = graphics_GetWidth(hw) / 2;
y = graphics_GetHeight(hw) / 2;

graphics_Text(hw, x, y,"hello world");
}
```

Whenever the user takes an action that requires the inset's window to be redrawn, the inset's FullUpdate routine will be called. The routine should do whatever it must do to redraw its part of the display. In this example, when *helloworld_FullUpdate* gets called, we would like it to draw the string *hello world* in the center of the window.

Understanding how to do this requires understanding a little more about the structure of insets. For a complete discussion of the inset structure, see *Inset Routines* (p. 91).

Recall that an inset is simply a rectangle of pixels in a window. Graphics calls are relative to insets rectangle, so we only need to divide the inset's height and width by 2 to calculate the coordinates for drawing the text at the midpoint of the inset's rectangle. *graphics_GetWidth(hw)* gets the inset's width and *graphics_GetHeight(hw)*, the height.

The next statement, *graphics_Text*, actually draws *hello world* in the window.

You must provide an *x_FullUpdate (xp, how)* routine for any inset *x* that you want to create. Basically, the routine should fully update its drawing in its rectangle. Upon the first update call, it must create the drawing; on other update calls it must redraw the drawing. Later examples will introduce more complexities in the full update procedures (See also *Inset Routines*, p. 91).

Exporting and importing routines

The program must export its inset driver routines. To do so, it must declare the routines for export and then export them. The program must import all the BE2 routines. Each file must also import any other external procedures it uses.

Declaring the driver routines for export

To declare the driver routines for export, you will use three macros, *BeginModule*, *EndModule* and *Entry*. *BeginModule* and *EndModule* identifies the module to the BE2 linking facilities, declares the number of exported procedures, and delimits the extent of the declaration. *Entry* gives the name and the return type of each exported procedure. Since the .h header file typically must be #included in each .c file for a module, it is a good place to declare exports. Thus, in Example 1, *helloworld.h* will contain the following declaration:

```
BeginModule(helloworld, 3)
    Entry(helloworld_New, struct helloworld *)
    Entry(helloworld_Init, int)
    Entry(helloworld_FullUpdate, int)
EndModule()
```

BeginModule defines the start of the module declaration. The first parameter is the name of the module, *helloworld,* and the second is the number of exported procedures--in this example, three. The next three lines use the *Entry* macro to declare the three driver routines for export from the module. Each call to *Entry* should give the name of the procedure and the type of value that it returns.

Importing BE2 routines and exporting driver routines

The file *helloworld.c*, the program file for the inset routines, begins by importing the constructs it needs from the BE2 library and exporting the driver routine declarations in the helloworld.h file.

```
#include "CamphorImport.h"
#include "be2/graphics.h"
#include "be2/inset.h"

#include "CamphorExport.h"
#include "helloworld.h"
```

The include files *CamphorImport.h* and *CamphorExport.h* define different versions of *BeginModule*, *EndModule* and *Entry* macros. *CamphorImport.h* defines the macros to produce the code required to import procedures from another module during run-time. *CamphorExport.h* defines the macros to produce the code required to export procedures from a module.

For this example, *helloworld.c*, the inset driver file, must #include *be2/inset.h* to access the BE2 inset facilities, and export *helloworld.h.*, the file containing the declaration of exported procedures. The file also #includes *be2/graphics.h* to access the underlying window management facilities.

The file *main.c* must import the *be2/inset.h* to access the BE2 facilities, *be2/im.h* to access the inset/window management facilities,and *helloworld.h* to access the *helloworld* inset declarations, as follows:

```
#include "CamphorImport.h"
#include "be2/inset.h"
#include "be2/im.h"
#include "helloworld.h"
```

Note that the order of the libraries is important: inset.h must appear before im.h before helloworld.h.

Setting up an inset as a stand-alone application program

```
main()  {
    im_WindowPtr win;
    struct helloworld *hw;

    camphorinit(0, 0, 0, "/usr/andrew/lib/be2");
    staticload("im", imCamphorInitializer());
    staticload("inset", insetCamphorInitializer());
    staticload("helloworld",helloworldCamphorInitializer());

    hw = (struct helloworld *) inset_New("helloworld", NULL);
    win = im_CreateWindow(NULL);
    im_FillWindow(win, hw);

    im_KeyboardProcessor();
}
```

To set up an inset as a stand-alone application program, you should create a main program that declares information needed for linking and loading, creates an instance of the inset, creates a window, puts the inset in the window, and enters an interaction loop.

In this example, *main()*, defined in main.c, begins with a call to *camphorinit* which sets up the BE2 linking facilities to look for imported procedures in /usr/andrew/lib/be2. The program should make calls to *staticload* for all facilities that the application knows it will need in advance, because the program will be significantly smaller and slightly faster.

The call to *inset_New* creates an instance of a *helloworld* inset. *im_CreateWindow* creates the window and *im_FillWindow* puts the inset in the window. *im_KeyboardProcessor* enters an interaction loop that mediates communication between a user and the application.

Compiling insets for static linking

```
CFLAGS = -g -DSTATICLINKING -I"/usr/andrew/include"

.SUFFIXES:  .o

helloworld.o: helloworld.c helloworld.h

helloworld: main.o helloworld.o
        cc  -g -o hw main.o helloworld.o  /usr/andrew/lib/libbe2.a\
              /usr/andrew/lib/libitc.a
```

The Makefile for a inset that will stand alone is like any other Makefile (see *make* in the online help pages), with the exception of the CFLAGS -DSTATICLINKING, which informs the BE2 dynamic linking facilities to generate code for static rather than dynamic linking.

To compile the program using this Makefile, you should have the files *helloworld.h*, *helloworld.c*, *main.c* and the *Makefile* itself in a single directory in which you have read, write and list permissions. You may copy these files from */usr/andrew/doc/be2/example1*.

Make the directory in which you have put the files the current directory. Then at the **Typescript** prompt, type

```
make helloworld
```

and press the Enter key.

To run after compilation, type

```
helloworld
```

and press the Enter key.

Program Listing for Example 1

helloworld.h

```
struct helloworld  {
    struct inset myinset; /* BE2 inset structure */
};

BeginModule(helloworld, 3)
    Entry(helloworld_New, struct helloworld *)
    Entry(helloworld_Init, int)
    Entry(helloworld_FullUpdate, int)
EndModule()
```

helloworld.c

```
#include "CamphorImport.h"
#include "be2/graphics.h"
#include "be2/inset.h"

#include "CamphorExport.h"
#include "helloworld.h"


struct helloworld *helloworld_New()  {
    register struct helloworld *hw;

    hw = (struct helloworld *) malloc(sizeof(struct helloworld));

    helloworld_Init(hw);

    return(hw);
}

helloworld_Init(hw)
struct helloworld *hw;  {
    inset_InitStructure("helloworld", hw);
}

helloworld_FullUpdate(hw, how)
struct helloworld *hw;
int how;  {

    int x,y;

    x = graphics_GetWidth(hw) / 2;
    y = graphics_GetHeight(hw)/ 2;

    graphics_Text(hw, x, y, "hello world");
}
```

main.c

```
#include "CamphorImport.h"
#include "graphics.h"
#include "be2/inset.h"
#include "be2/im.h"
#include "helloworld.h"

main()  {
    im_WindowPtr win;
    struct helloworld *hw;

    camphorinit(0, 0, 0, "/usr/andrew/lib/be2");
    staticload("im", imCamphorInitializer());
    staticload("inset", insetCamphorInitializer());
    staticload("helloworld",helloworldCamphorInitializer());

    hw = (struct helloworld *) inset_New("helloworld",NULL);
    win = im_CreateWindow(NULL);
    im_FillWindow(win, hw);

    im_KeyboardProcessor();
}
```

Makefile

```
CFLAGS = -g -DSTATICLINKING -I"/usr/andrew/include"

.SUFFIXES:  .o

helloworld.o: helloworld.c helloworld.h

helloworld: main.o helloworld.o
      cc  -g -o hw main.o helloworld.o  /usr/andrew/lib/libbe2.a
      /usr/andrew/lib/libitc.a
```

Example 2: Dynamically loading an inset

Example 1 described the basic program structure needed to work with insets in a stand alone application program. Example 2 will introduce the changes you need to make in order to build an inset that can be dynamically loaded into a multi-media editor or other BE2 application. Dynamic loading makes it possible for application programmers to develop their own insets, independent of the developers of the BE2 and other applications making use of BE2, while still allowing these new, user-defined insets to be used in any multi-media editor or other application with dynamic loading capabilities.

For this example, we will be modifying the *helloworld* inset created in Example 1.

Running the example program

Before reading the discussion of the example program, you may find it helpful to run the program on your workstation. The following gives direction for dynamically loading the inset into **Brand-X**, or **bx**, a multi-media editor developed using BE2.

Action 1. To set-up the dynamic loading subsystem's Dynamic Loading PATH, DLPATH, so that it finds the inset *helloworld* in /usr/andrew/doc/be2/example2, at the **Typescript** prompt, type

```
setenv DLPATH /usr/andrew/doc/be2/example2:/usr/andrew/lib/be2
```

and press the Enter key.

Response. The Typescript prompt will reappear. *setenv* DLPATH sets the environment variable DLPATH to the value */usr/andrew/doc/be2/example2:/usr/andrew/lib/be2* (See *csh (1)*). As a result, the dynamic loading subsystem will look for insets in both */usr/andrew/doc/be2/example2*, the location of inset for this example, and */usr/andrew/lib/be2*, the location of some already defined insets.

Action 2. To run **bx**, the multi-media editor, at the **Typescript** prompt, type

```
bx
```

and press the Enter key.

Response. You should get a **Brand-X** editor window on your screen.

Action 3. To load an inset, you will need some text to work with. To create text, move the cursor to the editor window and type in some characters. Three short lines of one or two words each will suffice.

Response. You should have a window that contains some text.

Action 4. To dynamically load the inset, type

ESC-X and do *not* press Enter.

Response. Brand-X will prompt you for with the word *Function:*

Action 5. At the prompt, type

```
vcmds_addinsetstyle
```

and press the Enter key.

Response. Brand-X will prompt you with the prompt, *Add style for inset:*

Action 6. At the prompt, type

```
helloworld
```

and press the Enter key.

Response. Brand-X will add a new menu card labeled *Inset* with menu item **helloworld** on the card.

Action 7. To create an instance of the inset, select a region of text, pop up the menus, and choose **helloworld** from the *Inset* menu.

Response. The *helloworld* inset will replace the text in the selected region. You can repeat Action 7 to create multiple instances of the inset.

Action 8. To quit the program, pop up the menus and choose **Quit** from the *Top* menu card.

Response. Brand-X will prompt you with the message, *Modified buffers exist, exit anyway*?

Action 9. At the prompt, type

```
y
```

and press the Enter key.

Response. The **Brand-X** window will disappear from the screen.

Creating the inset

The header file *helloworld.h* is exactly the same as in Example 1. The driver routines in *helloworld*.c are exactly the same except for the *helloworld_FullUpdate* routine. Recall that in Example 1, *helloworld_FullUpdate* was declared with two parameters, a pointer to the *helloworld* inset, *hw*, and an integer, *how*. But Example 1 ignored the *how* parameter. The *how* parameter specifies how the inset's environment has changed and can have the following values:

inset_FULLREDRAW -- the inset should be completely redrawn.
inset_PARTIALREDRAW - the inset should be redrawn within the rectangle specified by the parameter *r;* further partial redraws will follow.
inset_LASTPARTIALREDRAW - the inset should be redrawn within the rectangle specified by the parameter *r*; this is the last partial redraw.
inset_REMOVE - the inset in being removed from the screen.

Example 1 ignored the *how* parameter and always did a full redraw of the inset. But in a multi-media editing environment, the *helloworld* inset cannot assume that the *how* parameter will have the value inset_FULLREDRAW. Furthermore, it is only on an inset_FULLREDRAW that an inset may assume that its rectangle has been cleared by *inset_FullUpdate*. So we must modify the *helloworld* inset to test for whether it needs to clear its portion of the window before drawing. Note that the *helloworld_FullUpdate* can still ignore inset_PARTIALREDRAW and inset_REMOVE values, since it does not partially redraw its rectangle, nor does it do anything special upon being removed from view.

```
helloworld_FullUpdate(hw, how)
struct helloworld *hw;
int how;  {
    int x,y;

    if (how == inset_FULLREDRAW || how == inset_LASTPARTIALREDRAW)
       {
       if (how == inset_LASTPARTIALREDRAW)  {
           /* Have to clear out the rectangle since it is not
       cleared on a partial redraw */

           graphics_Clear (hw);
       }
       x = hw->myinset.r.width / 2;
       y = hw->myinset.r.height / 2;

      graphics_Text (hw, x, y, "hello world");
    }
}
```

The first *if* statement, *if (how == inset_FULLREDRAW || how ==*

inset_LASTPARTIALREDRAW), optimizes the *helloworld_FullUpdate* procedure, since the inset does not need to take any action upon a partial redraw or removal. The second *if* statement, *if (how == inset_LASTPARTIALREDRAW)*, tests to see whether the rectangle must be cleared, and if it does, *graphics_Clear (hw)* actually clears the inset *hw*'s rectangle.

The rest of the procedure is like Example 1--it calculates the coordinates for the center of the inset's rectangle and draws *hello world* there.

Compiling insets for dynamic loading

With static linking, the linker combines several object programs into a single program, searching libraries and resolving all external references. If there are no errors, the output of the linker is an executable file (see *ld (1)* in the Unix man or Andrew on-line help pages).

If STATICLINKING is defined, in the Makefile for the program, the procedure declarations are converted to extern type proc (). With dynamic linking, all procedure declarations are transformed to be illegal instruction traps. These are then handled by a signal handler. Thus, static linking is more efficient if you do not need the inset to be dynamically loaded.

If the inset you are defining *is* going to be dynamically loaded, you must include the following in your Makefile:

```
.SUFFIXES:  .6.o

o.6:

    make6 $@
```

make6 is a shell script, located in /usr/andrew/bin, that runs the linker to produce a relocatable object module. Relocatable object modules have their references to relative positions rather than absolute addresses. Relocatable modules are necessary in order for dynamic loading to work. Relative addresses are usually less efficient than absolute addresses, so if you are not going to dynamically load, you should still statically load as much as possible.

The complete Makefile looks like the following:

```
CFLAGS = -g -I"/usr/andrew/include"

.SUFFIXES: .6 .o

.o.6:
	make6 $*

all: helloworld helloworld.6

$@.6: $@.o

helloworld.o: helloworld.c helloworld.h


helloworld: main.o  helloworld.o
	cc -g -o  helloworld main.o helloworld.o\
		  /usr/andrew/lib/libbe2.a\
		  /usr/andrew/lib/libitc.a
```

Note that STATICLINKING is *not* specified, since dynamic linking is the default for BE2. As mentioned above, the *make6* procedure creates the .6 files for the inset, which are relocatable.

To compile the program using this *Makefile*, you should have the files *helloworld.h*, *helloworld.c*, *main.c* and the *Makefile* itself in a single directory in which you have read, write and list permissions. Note that you do not need *main.c* in order to make the *helloworld.6* module, but you still need it to make *helloworld*. Thus, if you were creating an inset that would never act as a stand alone application, you would not need *main.c*.

Make the directory in which you have put the files the current directory. Then at the **Typescript** prompt, type

```
make all
```

and press the Enter key.

The *Makefile* will generate *helloworld*, a stand alone inset, and *helloworld.6*, a dynamically loadable *helloworld* inset.

Program Listing for Example 2

helloworld.h

```
struct helloworld  {
    struct inset myinset; /* inset structure */
};

BeginModule(helloworld, 3)
    Entry(helloworld_New, struct helloworld *)
    Entry(helloworld_Init, int)
    Entry(helloworld_FullUpdate, int)
EndModule()
```

helloworld.c

```
#include "CamphorImport.h"
#include "be2/graphics.h"
#include "be2/inset.h"

#include "CamphorExport.h"
#include "helloworld.h"

struct helloworld *helloworld_New()  {
    register struct helloworld *hw;

    hw = (struct helloworld *) malloc(sizeof(struct helloworld));

    helloworld_Init(hw);

    return(hw);
}

helloworld_Init(hw)
struct helloworld *hw;  {
    inset_InitStructure("helloworld", hw);
}

helloworld_FullUpdate(hw, how)
struct helloworld *hw;
int how;  {
    int x,y;

    if (how == inset_FULLREDRAW || how == inset_LASTPARTIALREDRAW)  {
      if (how == inset_LASTPARTIALREDRAW)  {
          /* Have to clear out the rectangle since it is not cleared on a
      partial redraw */

          graphics_Clear (hw);
      }
      x = graphics_GetWidth(hw) / 2;
```

```
        y = graphics_GetHeight(hw) / 2;

        graphics_Text (hw, x, y, "hello world");
    }
}
```

main.c

```
#include "CamphorImport.h"
#include "be2/inset.h"
#include "be2/im.h"
#include "helloworld.h"

main()  {
    im_WindowPtr win;
    struct helloworld *hw;

    camphorinit(0, 0, 0, "/usr/andrew/lib/be2");
    staticload("im", imCamphorInitializer());
    staticload("inset", insetCamphorInitializer());
    staticload("helloworld",helloworldCamphorInitializer());

    hw = (struct helloworld *) inset_New("helloworld", 0);
    win = im_CreateWindow(0);
    im_FillWindow(win, hw);

    im_KeyboardProcessor();
}
```

Makefile

```
CFLAGS = -g -I"/usr/andrew/include"

.SUFFIXES: .6 .o

.o.6:
        make6 $*

all: helloworld helloworld.6

$@.6: $@.o

helloworld.o: helloworld.c helloworld.h

helloworld: main.o  helloworld.o
        cc -g -o  helloworld main.o helloworld.o\
                  /usr/andrew/lib/libbe2.a\
                  /usr/andrew/lib/libitc.a
```

Example 3: Responding to mouse hits

Example 1 described the basic program structure for an inset, and Example 2 showed how to dynamically load the inset into a multi-media editor. This section will show how to take the inset in Example 2, and modify it into a inset that can be moved around in a window in response to mouse hits.

Running the example program

Before reading the discussion of the example program, you may find it helpful to run the program.

Action 1. To run the program, at the **Typescript** prompt, type

/usr/andrew/doc/be2/example3/helloworld

and press the Enter key.

Response. The program will produce a window with *hello world* centered in the body.

Action 2. Re-shape the program's window.

Response. The inset will respond to an update request and redraw *hello world* in the center.

Action 3. Click with the left mouse button.

Response. The program will respond to an update request and redraw *hello world* at the position of the mouse cursor at the time of the click.

Action 4. To quit the program, move the cursor to the window title bar, pop-up the menus and choose **Zap** from the card labeled *This Window*.

Response. The window will disappear from the screen.

Declaring a structure for an inset

For a discussion on structure declaration, see Example 1.

Deciding on a name for the inset

For this example, we will continue to use the inset name, *helloworld*.

Declaring the data structure for the inset

The data structure for the inset will still be declared in the file *helloworld.h*, but will be slightly different from the declaration in Example 1 in that some data specific to the inset must be defined.

```
struct helloworld  {
    struct inset myinset;
    int x,y;              /* current location of the string */
    int newx, newy;       /* new position for the string */
};
```

The integer variables *x* and *y* hold the coordinates of the current location of the string *hello world* in the window. *newx* and *newy* are defined by the position of the mouse cursor at the time of a mouse button click, and tell the inset the position in the window *hello world* should be redrawn.

Defining the driver routines for the inset

The following discussion illustrates how to define the inset driver routines for the inset. For a complete discussion of driver routines, refer to *Example 1*. Again, not all the possible inset driver routines are used in this example; the defaults are used instead.

Example 3 requires five driver routines: a routine to create instances of the inset, a routine to initialize an instance of the inset, and a routine to respond to requests to do a full update of the inset's drawing, a routine to do a simple update of the drawing, and a routine to record mouse hits.

Creating the inset

```
struct helloworld *helloworld_New()  {
struct helloworld *hw;

    hw = (struct helloworld *) malloc(sizeof(struct helloworld));

    helloworld_Init(hw);

    return(hw);
}
```

The creation routine *helloworld_New* is the same as in the previous examples.

Initializing the inset

```
helloworld_Init(hw)
struct helloworld *hw;  {
    inset_InitStructure("helloworld", hw);
    hw->x = hw->y = helloworld_BADPOS;
    hw->newx = hw->newy = helloworld_BADPOS;
}
```

This is the same as in previous examples except for the last two lines, which act to guarantee that the x,y and *newx*, *newy* are not being displayed somewhere else at the time of initialization by assigning them the value helloworld_BADPOS, the largest negative number available on a (32 bit) machine.

Full update request

```
helloworld_FullUpdate(hw, how)
struct helloworld *hw;
int how;  {

    if (how == inset_FULLREDRAW || how == inset_LASTPARTIALREDRAW)
       {
       if (how == inset_LASTPARTIALREDRAW)  {

           graphics_Clear(hw);
       }

       if (hw->x == helloworld_BADPOS)  {
           hw->newx = hw->x = graphics_GetWidth(hw) / 2;
           hw->newy = hw->y = graphics_GetHeight(hw) / 2;
       }

       graphics_Text(hw, hw->x, hw->y, "hello world");
    }
}
```

This is the same as in Example 2, except that the coordinates of the mouse hit, *newx* and *newy*, are also calculated, and the text string is redrawn at that point when the left mouse button is clicked.

A request for an update

```
helloworld_Update(hw)
struct helloworld *hw;  {
    if (hw->newx != hw->x || hw->newy != hw->y)  {
       graphics_SetFunction(hw, graphics_GCinvert);
       graphics_Text(hw, x, y, "hello world");
       hw->x = hw->newx;
       hw->y = hw->newy;
       graphics_Text(hw, x, y, "hello world");
       graphics_SetFunction(hw, graphics_GCset);
    }
}
```

The difference between an update and a full update is that an update merely redraws a specified portion of the inset, whereas a full update redraws everything in the entire window. So, if the window is changed, then the inset will full update, but if there is just an input from the mouse, and no other change, then the inset just does a simple update, and redraws only clears the text string at the old position and draws at the new position.

The first graphics_SetFunction in this example is telling the inset that if the screen is white, print the text in black, and vice versa (graphics_GCinvert); the second sets the conditions back to the original state.

Handling mouse input

```
struct inset *helloworld_Hit(hw, action, x, y)
struct helloworld *hw;
int action, x, y;  {
    if (action == MouseMask(LeftButton, DownTransition) || action
       == MouseMask(RightButton, DownTransition))  {
       hw->newx = x;
       hw->newy = y;
       inset_WantUpdate(hw, hw);
    }

    return (struct inset *) hw;
}
```

This procedure handles the mouse input for the inset. When the left mouse button is hit (DownTransition), newx,newy is calculated and made the new "current" position for the text string. Then, the procedure requests an update so that the string can be redrawn in the new position.

Exporting and importing routines

For a complete description of exporting driver routines and importing BE2 routines, see Example 1.

Declaring the driver routines for export

In this example, *helloworld.h* will contain the following declaration:

```
BeginModule(helloworld, 5)
    Entry(helloworld_New, struct helloworld *)
    Entry(helloworld_Init, int)
    Entry(helloworld_FullUpdate, int)
    Entry(helloworld_Update, int)
    Entry(helloworld_Hit, struct inset *)
EndModule()

#define helloworld_BADPOS 0x80000000
```

Importing BE2 routines and exporting driver routines

The declarations are exactly the same as in the first two examples:

```
#include "CamphorImport.h"
#include "be2/graphics.h"
#include "be2/inset.h"

#include "CamphorExport.h"
#include "helloworld.h"
```

Similarly, for the file *main.c*, the declarations should be:

```
#include "CamphorImport.h"
#include "be2/inset.h"
#include "be2/im.h"
#include "helloworld.h"
```

Setting up the inset

```
main()  {
    im_WindowPtr win;
    struct helloworld *hw;

    camphorinit(0, 0, 0, "/usr/andrew/lib/be2");
    staticload("im", imCamphorInitializer());
    staticload("inset", insetCamphorInitializer());
    staticload("helloworld",helloworldCamphorInitializer());

    hw = (struct helloworld *) inset_New("helloworld", NULL);
    win = im_CreateWindow(NULL);
    im_FillWindow(win, hw);

    im_KeyboardProcessor();
}
```

The body of the *main.c* file is exactly the same as in Example 1.

Compiling the inset

```
CFLAGS = -g -DSTATICLINKING -I"/usr/andrew/include"

.SUFFIXES:  .o

helloworld.o: helloworld.c helloworld.h

helloworld: main.o helloworld.o
        cc  -g -o hw main.o helloworld.o  /usr/andrew/lib/libbe2.a\
              /usr/andrew/lib/libitc.a
```

The Makefile for this example is exactly as in Example 1, and is a static link compilation. For a detailed description and instructions on compiling, see the discussion in Example 1.

Program Listing for Example 3

helloworld.h

```
struct helloworld  {
    struct inset myinset; /* inset structure */
    int x,y;              /* current location of the string */
    int newx, newy;       /* new position for the string */
};

BeginModule(helloworld, 5)
    Entry(helloworld_New, struct helloworld *)
    Entry(helloworld_Init, int)
    Entry(helloworld_FullUpdate, int)
    Entry(helloworld_Update, int)
    Entry(helloworld_Hit, struct inset *)
EndModule()

#define helloworld_BADPOS 0x80000000
```

helloworld.c

```
#include "CamphorImport.h"
#include "be2/graphics.h"
#include "be2/inset.h"

#include "CamphorExport.h"
#include "helloworld.h"

struct helloworld *helloworld_New()  {
    register struct helloworld *hw;

    hw = (struct helloworld *) malloc(sizeof(struct helloworld));

    helloworld_Init(hw);

    return(hw);
}

helloworld_Init(hw)
struct helloworld *hw;  {
    inset_InitStructure("helloworld", hw);
    hw->x = hw->y = helloworld_BADPOS;
    hw->newx = hw->newy = helloworld_BADPOS;
}
```

```
helloworld_FullUpdate(hw, how)
struct helloworld *hw;
int how;  {

    if (how == inset_FULLREDRAW || how == inset_LASTPARTIALREDRAW)  {
      if (how == inset_LASTPARTIALREDRAW)  {
/*        Have to clear out the rectangle since it is not cleared on a
      partial redraw */

          graphics_Clear(hw);
      }

      if (hw->x == helloworld_BADPOS)  {
          hw->newx = hw->x = graphics_GetWidth(hw) / 2;
          hw->newy = hw->y = graphics_GetHeight(hw) / 2;
      }

      graphics_Text(hw, x, y, "hello world");
    }
}

helloworld_Update(hw)
struct helloworld *hw;  {
    if (hw->newx != hw->x || hw->newy != hw->y)  {
      graphics_SetFunction(hw, graphics_GCinvert);
      graphics_Text(hw, x, y, "hello world");
      hw->x = hw->newx;
      hw->y = hw->newy;
      graphics_Text(hw, x, y, "hello world");
      graphics_SetFunction(hw, graphics_GCset);
    }
}

struct inset *helloworld_Hit(hw, action, x, y)
struct helloworld *hw;
int action, x, y;  {
    if (action == MouseMask(LeftButton, DownTransition) || action ==
      MouseMask(RightButton, DownTransition))  {
      hw->newx = x;
      hw->newy = y;
      inset_WantUpdate(hw, hw);
    }

    return (struct inset *) hw;
}
```

main.c

```
#include "CamphorImport.h"
#include "be2/inset.h"
#include "be2/im.h"
#include "helloworld.h"

main()  {
    im_WindowPtr win;
    struct helloworld *hw;

    camphorinit(0, 0, 0, "/usr/andrew/lib/be2");
    staticload("im", imCamphorInitializer());
    staticload("inset", insetCamphorInitializer());
    staticload("helloworld",helloworldCamphorInitializer());

    hw = (struct helloworld *) inset_New("helloworld", 0);
    win = im_CreateWindow(0);
    im_FillWindow(win, hw);

    im_KeyboardProcessor();
}
```

Makefile

```
CFLAGS = -g -DSTATICLINKING -I"/usr/andrew/include"

.SUFFIXES:  .o

helloworld.o: helloworld.c helloworld.h

helloworld: main.o helloworld.o
      cc  -g -o hw main.o helloworld.o  /usr/andrew/lib/libbe2.a\
            /usr/andrew/lib/libitc.a
```

Example 4: Working with menus

Example 3 showed how to move an inset around in a window. This example will show how to add menus to an inset, by modifying the *helloworld* inset of Example 3.

Running the example program

Before reading the discussion of the example program, you may find it helpful to run the program.

Action 1. To run the program, at the **Typescript** prompt, type

/usr/andrew/doc/be2/example4/helloworld

and press the Enter key.

Response. The program will produce a window with *hello world* centered in the body.

Action 2. Re-shape the program's window.

Response. The inset will respond to an update request and redraw *hello world* in the center.

Action 3. Click with the left mouse button.

Response. The program will produce *hello world* in the window at the mouse cursor position at the time of the click.

Action 4. Click both mouse buttons at once.

Response. You will get a pop-up menu for the inset with two menu items: *Center* and *Inverse*. If you choose *Center*, hello world will be redrawn at the center of the window; if you choose *Invert*, the inset rectangle will turn black, with hello world drawn in white letters. Choosing *Invert* again will flip things back.

Action 4. To quit the program, move the cursor to the window title bar, pop-up the menus and choose **Zap** from the card labeled *This Window*.

Response. The window will disappear from the screen.

Declaring a structure for an inset

For a discussion on structure declaration, see Example 1.

Deciding on a name for the inset

For this example, we will continue to use the inset name, *helloworld*.

Declaring the data structure for the inset

```
struct helloworld  {
    struct inset myinset; /* inset structure */
    int x,y;              /* current location of the string */
    int newx, newy;       /* new position for the string */
    char blackonwhite;     /* true if text is to be drawn black on
      white */
    char newblackonwhite;
    struct menuList *menus; /* menus for this inset */
};
```

This is the same as in previous example with the addition of variables (*blackonwhite*, *newblackonwhite*) that the inset will use to tell what state the inset rectangle is in (True if background is white), as well as a menuList structure, which will contain the menus and menu items for the inset.

Defining the driver routines for the inset

Example 4 requires the same five driver routines from Example 3: a routine to create instances of the inset, a routine to initialize an instance of the inset, and a routine to respond to requests to do a full update of the inset's drawing, a routine to do a simple update of the drawing, and a routine to record mouse hits. No extra routines are necessary for the program to work.

Providing a procedure for string centering

```
centerstring(hw)
struct helloworld *hw;  {
    hw->newx = graphics_GetWidth(hw) / 2;
    hw->newy = graphics_GetHeight(hw) / 2;
    inset_WantUpdate(hw, hw);
}
```

This example, however, does require two extra procedures, the first of which computes the center of the window and requests that the text be redrawn there. In the previous example, this was done only in the FullUpdate routine.

Providing a procedure to make the inset invert

```
invertbackground(hw)
struct helloworld *hw;  {
    hw->newblackonwhite = ! hw->newblackonwhite;
    inset_WantUpdate(hw, hw);
}
```

The second additional procedure keeps track of the current state and new state of the window (black on white or white on black) and requests that the inset be redrawn accordingly.

Creating the inset

```
struct helloworld *helloworld_New()  {
    struct helloworld *hw;

    hw = (struct helloworld *) malloc(sizeof(struct helloworld));

    helloworld_Init(hw);

    return(hw);
}
```

This is still the same as in previous examples.

Initializing the inset

```
helloworld_Init(hw)
struct helloworld *hw;  {
    inset_InitStructure("helloworld", hw);
    hw->x = hw->y = helloworld_BADPOS;
    hw->newx = hw->newy = helloworld_BADPOS;
    hw->newblackonwhite = hw->blackonwhite = 1;
    hw->menus = im_NewML();
    im_AddToML(hw->menus, centerstring, "Center", hw, 0);
    im_AddToML(hw->menus, invertbackground, "Invert", hw, 0);
}
```

The difference between the initialization procedure for this example and Example 3 is that *newblackonwhite* and *blackonwhite* are initialized and set to 1, and the menus are created to have the menu items *Center* and *Invert*. Menus are always added to a "menu list," so a new menu list is created first, then the menu for the inset is added to it.

Full update request

```
helloworld_FullUpdate(hw, how)
struct helloworld *hw;
int how;  {
    int x,y;

    if (how == inset_FULLREDRAW || how == inset_LASTPARTIALREDRAW)
       {
       if (how == inset_LASTPARTIALREDRAW)  {

           if (hw->blackonwhite)  {
              graphics_SetBackground(hw, graphics_WhitePixel);
           }
           else  {
              graphics_SetBackGround(hw, graphics_BlackPixel);
           }
           graphics_Clear(hw);
       }
       else if (! hw->blackonwhite) {
           /* Have to paint the rectangle black since it is set to
       white before a full redraw */

           graphics_SetBackGround(hw, BlackPixel);
           graphics_Clear(hw);
       }

       graphics_SetFunction(hw, graphics_GCinvert);

       if (hw->x == helloworld_BADPOS)  {
           hw->newx = hw->x = graphics_GetWidth / 2;
           hw->newy = hw->y = graphics_GetHeight / 2;
       }

       graphics_Text(hw, x, y, "hello world");

       inset_WantMenuList(hw, hw, hw->menus);
    }
}
```

The FullUpdate procedure for this example handles redraws for the two menu items. For *Invert*, if the current rectangle is white, then the it is painted black, and the characters drawn in white. If the rectangle is black, then the it is changed back to white, with black characters. For Center, the center of the rectangle is computed, and the text is redrawn at that point. If Center is chosen when the rectangle is black, then the rectangle must be repainted black, since the FullUpdate sets the rectangle to white before a full redraw.

A request for an update

```
helloworld_Update(hw)
struct helloworld *hw;  {

    if (hw->newblackonwhite != hw->blackonwhite)  {
       graphics_SetFunction(hw, graphics_GCinvert);
       graphics_FillRectangle(hw, graphics_GetWidth(hw),
       graphics_GetHeight(hw));
       hw->blackonwhite = hw->newblackonwhite;
    }
    if (hw->newx != hw->x || hw->newy != hw->y)  {
       graphics_SetFunction(hw, graphics_GCinvert);
       graphics_Text(hw, x, y, "hello world");
       hw->x = hw->newx;
       hw->y = hw->newy;
       graphics_Text(hw, x, y, "hello world");
       graphics_SetFunction(hw, graphics_GCset);
    }
}
```

The first part of this procedure handles updates that require redrawing the inset in inverse mode. Based on the values of *newblackonwhite* and *blackonwhite* passed to it, the update procedure will redraw in the mode opposite the current mode, filling in the entire visible rectangle as either white or black.

The second part of the procedure is exactly the same as in Example 3, and handles redrawing of the text string according to mouse hit information.

Handling mouse input

```
struct inset *helloworld_Hit(hw, action, x, y)
struct helloworld *hw;
int action, x, y;  {
    if (action == MouseMask(LeftButton, DownTransition) || action
       == MouseMask(RightButton, DownTransition))  {
       hw->newx = x;
       hw->newy = y;
       inset_WantUpdate(hw, hw);
    }

    return (struct inset *) hw;
}
```

This procedure is exactly the same as in Example 3.

Exporting and importing routines

For a complete description of exporting driver routines and importing BE2 routines, see Example 1.

Declaring the driver routines for export

In this example, *helloworld.h* will contain the following declaration:

```
BeginModule(helloworld, 5)
    Entry(helloworld_New, struct helloworld *)
    Entry(helloworld_Init, int)
    Entry(helloworld_FullUpdate, int)
    Entry(helloworld_Update, int)
    Entry(helloworld_Hit, struct inset *)
EndModule()

#define helloworld_BADPOS 0x80000000
```

This is exactly the same as in the previous example, because no new routines are necessary.

Importing BE2 routines and exporting driver routines

In this example, the *helloworld.c* file should begin with the following declarations, which are exactly the same as in the previous examples:

```
#include "CamphorImport.h"
#include "be2/graphics.h"
#include "be2/inset.h"

#include "CamphorExport.h"
#include "helloworld.h"
```

Similarly, for the file *main.c*, the declarations should be:

```
#include "CamphorImport.h"
#include "be2/inset.h"
#include "be2/im.h"
#include "helloworld.h"
```

Setting up the inset

```
main()  {
    im_WindowPtr win;
    struct helloworld *hw;

    camphorinit(0, 0, 0, "/usr/andrew/lib/be2");
    staticload("im", imCamphorInitializer());
    staticload("inset", insetCamphorInitializer());
    staticload("helloworld",helloworldCamphorInitializer());

    hw = (struct helloworld *) inset_New("helloworld", NULL);
    win = im_CreateWindow(NULL);
    im_FillWindow(win, hw);

    im_KeyboardProcessor();
}
```

The body of the *main.c* file is exactly the same as in the previous examples.

Compiling the inset

```
CFLAGS = -g -DSTATICLINKING -I"/usr/andrew/include"

.SUFFIXES:  .o

helloworld.o: helloworld.c helloworld.h

helloworld: main.o helloworld.o
      cc  -g -o hw main.o helloworld.o
      /usr/andrew/lib/libbe2.a\
            /usr/andrew/lib/libitc.a
```

The Makefile for this example is exactly as in the previous examples, and is a static link compilation. For a detailed description and instructions on compiling, see the discussion in Example 1.

Program Listing for Example 4

helloworld.h

```
struct helloworld  {
    struct inset myinset; /* inset structure */
    int x,y;              /* current location of the string */
    int newx, newy;       /* new position for the string */
    char blackonwhite;    /* true of text is to be drawn black on white */
    char newblackonwhite;
    struct menuList *menus; /* menus for this inset */
};

BeginModule(helloworld, 5)
    Entry(helloworld_New, struct helloworld *)
    Entry(helloworld_Init, int)
    Entry(helloworld_FullUpdate, int)
    Entry(helloworld_Update, int)
    Entry(helloworld_Hit, struct inset *)
EndModule()

#define helloworld_BADPOS 0x80000000
```

helloworld.c

```
#include "CamphorImport.h"
#include "be2/graphics.h"
#include "be2/inset.h"

#include "CamphorExport.h"
#include "helloworld.h"

centerstring(hw)
struct helloworld *hw;  {
    hw->newx = GetWidth(hw) / 2;
    hw->newy = GetHeigth(hw) / 2;
    inset_WantUpdate(hw, hw);
}

invertbackground(hw)
struct helloworld *hw;  {
    hw->newblackonwhite = ! hw->newblackonwhite;
    inset_WantUpdate(hw, hw);
}
```

```
struct helloworld *helloworld_New()  {
    register struct helloworld *hw;

    hw = (struct helloworld *) malloc(sizeof(struct helloworld));

    helloworld_Init(hw);

    return(hw);
}

helloworld_Init(hw)
struct helloworld *hw;  {
    inset_InitStructure("helloworld", hw);
    hw->x = hw->y = helloworld_BADPOS;
    hw->newx = hw->newy = helloworld_BADPOS;
    hw->newblackonwhite = hw->blackonwhite = 1;
    hw->menus = im_NewML();
    im_AddToML(hw->menus, centerstring, "Center", hw, 0);
    im_AddToML(hw->menus, invertbackground, "Invert", hw, 0);
}

helloworld_FullUpdate(hw, how)
struct helloworld *hw;
int how;  {
    int x,y;

    if (how == inset_FULLREDRAW || how == inset_LASTPARTIALREDRAW)  {
      if (how == inset_LASTPARTIALREDRAW)  {
          /* Have to clear out the rectangle since it is not cleared on a
      partial redraw */

          if (hw->blackonwhite)  {
             graphics_SetBackground(hw, graphics_WhitePixel)
          }
          else  {
             graphics_SetBackGround(hw, graphics_BlackPixel);
          }
          graphics_Clear(hw);
      }
      else if (! hw->blackonwhite) {
          /* Have to paint the rectangle black since it is set to white
      before a full redraw */

          graphics_SetBackGround(hw, BlackPixel);
          graphics_Clear(hw);
      }

      graphics_SetFunction(hw, graphics_GCinvert);

      if (hw->x == helloworld_BADPOS)  {
          hw->newx = hw->x = GetWidth(hw) / 2;
```

```
            hw->newy = hw->y = GetHeight(hw)/ 2;
        }

        graphics_Text(hw, x, y, "hello world");

        inset_WantMenuList(hw, hw, hw->menus);
    }
}

helloworld_Update(hw)
struct helloworld *hw;  {

    if (hw->newblackonwhite != hw->blackonwhite)  {
        graphics_SetFunction(hw, graphics_GCinvert);
        graphics_FillRectangle(hw, GetWidth(hw), GetHeight(hw));
        hw->blackonwhite = hw->newblackonwhite;
    }
    if (hw->newx != hw->x || hw->newy != hw->y)  {
        graphics_SetFunction(hw, graphics_GCinvert);
        graphics_Text(hw, x, y, "hello world");
        hw->x = hw->newx;
        hw->y = hw->newy;
        graphics_Text(hw, x, y, "hello world");
        graphics_SetFunction(hw, graphics_GCset);
    }
}

struct inset *helloworld_Hit(hw, action, x, y)
struct helloworld *hw;
int action, x, y;  {
    if (action == MouseMask(LeftButton, DownTransition) || action ==
        MouseMask(RightButton, DownTransition))  {
        hw->newx = x;
        hw->newy = y;
        inset_WantUpdate(hw, hw);
    }

    return (struct inset *) hw;
}
```

main.c

```
#include "CamphorImport.h"
#include "be2/inset.h"
#include "be2/im.h"
#include "helloworld.h"

main()  {
    im_WindowPtr win;
    struct helloworld *hw;

    camphorinit(0, 0, 0, "/usr/andrew/lib/be2");
    staticload("im", imCamphorInitializer());
    staticload("inset", insetCamphorInitializer());
    staticload("helloworld",helloworldCamphorInitializer());

    hw = (struct helloworld *) inset_New("helloworld", 0);
    win = im_CreateWindow(0);
    im_FillWindow(win, hw);

    im_KeyboardProcessor();
}
```

Makefile

```
CFLAGS = -g -DSTATICLINKING -I"/usr/andrew/include"

.SUFFIXES:  .o

helloworld.o: helloworld.c helloworld.h

helloworld: main.o helloworld.o
      cc  -g -o hw main.o helloworld.o  /usr/andrew/lib/libbe2.a\
            /usr/andrew/lib/libitc.a
```

Example 5: Responding to Keyboard Input

Example 4 described how to add menus to the inset. This example introduces the changes you need to make in order to build an inset that responds to keyboard input. It will define two keys: *ctrl-c* and *ctrl-i*. *ctrl-c* will center hello world in the window. *ctrl-i* will invert the inset's rectangle.

Running the example program

Before reading the discussion of the example program, you may find it helpful to run the program on your workstation.

Action 1. To run the program on your workstation, at the **Typescript** prompt, type

```
/usr/andrew/doc/example5/helloworld
```

and press the Enter key.

Response. The program will produce a window with *hello world* centered in the inset's rectangle,; in this case, in the body of the window..

Action 2. To move *hello world* in the window, position the mouse cursor within the program's window and click on the left mouse button.

Response. *hello world* will be drawn, centered on the point where you clicked the left mouse button.

Action 3. To use the keyboard to center *hello world* in the center of the inset's rectangle, type

```
ctrl-c
```

Response. *hello world* will be drawn in the center of the inset's rectangle.

Action 4. To invert the drawing, type

```
ctrl-i
```

Response. The background will invert. In this case, it will change from white to black. Action 3 and 4 can be repeated in any order.

Action 5. To quit the program, move your mouse cursor into the window's title bar, pop up the menus and choose **Zap** from the *This Window* menu card.

Response. The window will disappear from the screen.

Creating the inset

Adding the capability of responding to input from the keyboard requires defining a new procedure, *helloworld_KeyIn*. In general, if you are defining an inset named *x*, and you want it to be able to respond to keyboard input, you must provide a routine *x_KeyIn*.

Adding a routine to the header file exports

To add a new BE2 interface procedure to the inset, we must increase the number of exported procedures in the *BeginModule* statement from 5 to 6 and add *helloworld_KeyIn* to the *Entry* statements in the header file, *helloworld.h* as follows:

```
BeginModule(helloworld, 6)
    Entry(helloworld_New, struct helloworld *)
    Entry(helloworld_Init, int)
    Entry(helloworld_FullUpdate, int)
    Entry(helloworld_Update, int)
    Entry(helloworld_Hit, struct inset *)
    Entry(helloworld_KeyIn, int)
EndModule()
```

The rest of the header file remains exactly the same as in the previous example.

Defining a routine to respond to keyboard input

The driver routines in *helloworld*.c are exactly the same except for the addition of the following *helloworld_KeyIn* procedure:

```
int helloworld_KeyIn(hw, c)
struct helloworld *hw;
int c;  {
    if (c == 03)        /* Control C */
      centerstring(hw);
    else if (c == 011) /* Control I */
      invertbackground(hw);
    else
      return inset_KEYUNACCEPTABLE;
    return inset_KEYACCEPTABLE;
}
```

helloworld_KeyIn examines the character c to see whether it is a *ctrl-c* or a *ctrl-i*. If the character is a *ctrl-c*, it calls the routine *centerstring*, to center *hello world* in its' rectangle, then it returns the value inset_KEYACCEPTABLE to indicate that it has accepted a key. If the character is a a *ctrl-i*, it calls the routine *invertbackground* to invert the background of the rectangle, then it returns inset_KEYACCEPTABLE. Otherwise, it returns inset_KEYUNACCEPTABLE to indicate that it is not interested in any other keys.

For any inset *x*, its *x_KeyIn* procedure must be declared with two parameters: *ip*, a pointer to the inset, and *ch*, the character which has been sent to the inset. An inset will receive characters when it is the current input focus. An *x_KeyIn* routine should parse any sequence of characters it is sent and return the following:

inset_KEYACCEPTABLE -- if it accepts the key.
inset_KEYUNACCEPTABLE -- if the key is unacceptable.
inset_KEYPARTIALACCEPT -- if the the key is possibly acceptable, i.e., if it is part of a keystroke command sequence .

Finally, an *x_KeyIn* routine should test whether the value of *ch* is inset_KEYSTATERESET. If the value of *ch* is inset_KEYSTATERESET, then the *x_KeyIn* routine should reset it's parse state to the beginning of a command sequence. It is a way to set up a cancel mechanism to allow the user to cancel out of a keystroke command sequence. Of course, if an inset does not handle sequences of keystrokes (and thus never returns a value of inset_KEYPARTIALACCEPT), it can ignore values of inset_KEYSTATERESET for *ch*. For example, *helloworld_KeyIn* can ignore inset_KEYSTATERESET.

The *main.c* and *Makefile* remain the same.

Program Listing for Example 5

helloworld.h

```
struct helloworld  {
    struct inset myinset; /* inset structure */
    int x,y; /* current location of the string */
    int newx, newy; /* new position for the string */
    char blackonwhite; /* true of text is to be drawn black on white */
    char newblackonwhite;
    struct menuList *menus; /* menus for this inset */
};

BeginModule(helloworld, 6)
    Entry(helloworld_New, struct helloworld *)
    Entry(helloworld_Init, int)
    Entry(helloworld_FullUpdate, int)
    Entry(helloworld_Update, int)
    Entry(helloworld_Hit, struct inset *)
    Entry(helloworld_KeyIn, int)
EndModule()

#define helloworld_BADPOS 0x80000000
```

helloworld.c

```
#include "CamphorImport.h"
#include "be2/graphics.h"
#include "be2/inset.h"
#include "be2/im.h"

#include "CamphorExport.h"
#include "helloworld.h"

centerstring(hw)
struct helloworld *hw;  {
    hw->newx = graphics_GetWidth(hw)  / 2;
    hw->newy = graphics_GetHeight(hw) / 2;
    inset_WantUpdate(hw, hw);
}

invertbackground(hw)
struct helloworld *hw;  {
    hw->newblackonwhite = ! hw->newblackonwhite;
    inset_WantUpdate(hw, hw);
}
```

```
struct helloworld *helloworld_New()  {
register struct helloworld *hw;

    hw = (struct helloworld *) malloc(sizeof(struct helloworld));

    helloworld_Init(hw);

    return(hw);
}

helloworld_Init(hw)
struct helloworld *hw;  {
    inset_InitStructure("helloworld", hw);
    hw->x = hw->y = helloworld_BADPOS;
    hw->newx = hw->newy = helloworld_BADPOS;
    hw->newblackonwhite = hw->blackonwhite = 1;
    hw->menus = im_NewML();
    im_AddToML(hw->menus, centerstring, "Center", hw, 0);
    im_AddToML(hw->menus, invertbackground, "Invert", hw, 0);
}

helloworld_FullUpdate(hw, how)
struct helloworld *hw;
int how;  {

    if (how == inset_FULLREDRAW || how == inset_LASTPARTIALREDRAW)  {
      if (how == inset_LASTPARTIALREDRAW)  {
          /* Have to clear out the rectangle since it is not cleared on a
      partial redraw */

          if (hw->blackonwhite)  {
            graphics_SetBackground(hw, graphics_WhitePixel)
          }
          else  {
            graphics_SetBackGround(hw, graphics_BlackPixel);
          }
          graphics_Clear(hw);
      }
      else if (! hw->blackonwhite) {
          /* Have to paint the rectangle black since it is set to white
      before a full redraw */

          graphics_SetBackGround(hw, BlackPixel);
          graphics_Clear(hw);
      }

      graphics_SetFunction(hw, graphics_GCinvert);

      if (hw->x == helloworld_BADPOS)  {
          hw->newx = hw->x = graphics_GetWidth(hw)  / 2;
          hw->newy = hw->y = graphics_GetHeight(hw)  / 2;
```

```
        }

        graphics_Text(hw, x, y, "hello world");

        inset_WantMenuList(hw, hw, hw->menus);
    }
}

helloworld_Update(hw)
struct helloworld *hw;  {

    if (hw->newblackonwhite != hw->blackonwhite)  {
        graphics_SetFunction(hw, graphics_GCinvert);
        graphics_FillRectangle(hw, graphics_GetWidth(hw),
        graphics_GetHeight(hw));
        hw->blackonwhite = hw->newblackonwhite;
    }
    if (hw->newx != hw->x || hw->newy != hw->y)  {
        graphics_SetFunction(hw, graphics_GCinvert);
        graphics_Text(hw, x, y, "hello world");
        hw->x = hw->newx;
        hw->y = hw->newy;
        graphics_Text(hw, x, y, "hello world");
        graphics_SetFunction(hw, graphics_GCset);
    }
}

struct inset *helloworld_Hit(hw, action, x, y)
struct helloworld *hw;
int action, x, y;  {
    if (action == MouseMask(LeftButton, DownTransition) || action ==
        MouseMask(RightButton, DownTransition))  {
        hw->newx = x;
        hw->newy = y;
        inset_WantUpdate(hw, hw);
    }

    return (struct inset *) hw;
}

int helloworld_KeyIn(hw, c)
struct helloworld *hw;
int c;  {
    if (c == 03)                /* Control C */
        centerstring(hw);
    else if (c == 011)                  /* Control I */
        invertbackground(hw);
    else
        return inset_KEYUNACCEPTABLE;
    return inset_KEYACCEPTABLE;
}
```

main.c

```
#include "CamphorImport.h"
#include "be2/inset.h"
#include "be2/im.h"
#include "helloworld.h"

main()  {
    im_WindowPtr win;
    struct helloworld *hw;

    camphorinit(0, 0, 0, "/usr/andrew/lib/be2");
    staticload("im", imCamphorInitializer());
    staticload("inset", insetCamphorInitializer());
    staticload("helloworld",helloworldCamphorInitializer());

    hw = (struct helloworld *) inset_New("helloworld", 0);
    win = im_CreateWindow(0);
    im_FillWindow(win, hw);
    inset_WantInputFocus(hw, hw);

    im_KeyboardProcessor();
}
```

Makefile

```
CFLAGS = -g -DSTATICLINKING -I"/usr/andrew/include"

.SUFFIXES:  .o

helloworld.o: helloworld.c helloworld.h

helloworld: main.o helloworld.o
      cc  -g -o hw main.o helloworld.o  /usr/andrew/lib/libbe2.a
      /usr/andrew/lib/libitc.a
```

Example 6: Mapping keys to commands

The previous section introduced how to respond to keyboard input. This section describes how to use the keymap package in responding to keyboard input. The keymap package provides a set of facilities that will often make writing the *x_KeyIn* procedure for an inset *x* substantially easier. The keymap package allows you to map sequences of keyboard characters to procedures. Typically, each procedure will be command procedure, i.e., the procedure will interpret the sequence of characters that the user types as a command.

Running the example program

To the user, this program appears identical to Example 5: *ctrl-c* centers *hello world* and *ctrl-i* inverts the inset's rectangle. The difference resides solely in the underlying mechanisms for responding to the keyboard input.

Action1. To run the program, at the Typescript prompt, type

```
/usr/andrew/doc/be2/example6
```

and press the Enter key.

Response. The program will produce a window with *hello world* centered in the body of the window.

To proceed with running the porgram, proceed with *Action2* in *Example 5* (p. 52).

Declaring a *keymap* structure and a *keystate* structure

To use the keymap package, we must declare two structures: a keymap structure and a keystate structure. It is convenient to declare them in the *helloworld* structure in *helloworld.h*:

```
struct helloworld  {
    struct inset myinset;
    int x,y;
    int newx, newy;
    char blackonwhite;
    char newblackonwhite;
    struct menuList *menus;
    struct keymap *commands; /* map keys to commands */
    struct keystate *state; /* keep a keystate      */
};
```

Formally, a keymap is a function that maps a sequence of keyboard characters to a procedure. The keymap package allows you to ignore the actual implementation details. To understand how keymaps work, however, you may think of a keymap as an array with 128 entries, one for each ASCII character. Each entry in the keymap array may have one of three values: a procedure, another keymap, or a special value that indicates no binding.

If keymaps only mapped single keys to procedures, then there would be no need to keep track of the state of the mapping. To map *sequences* of keys to procedures, however, requires keeping state information. The keymap package uses a *keystate* structure to keep track of the state information.

Accessing the keymap library

```
#include "CamphorImport.h"
#include "be2/graphics.h"
#include "be2/inset.h"
#include "be2/im.h"
#include "be2/keymap.h"

#include "CamphorExport.h"
#include "helloworld.h"
```

To access the keymap package, we must *#include "be2/keymap.h"* in the files that use keymap routines. In this example, the files *helloworld.c* and *main.c*.

Creating and initializing the keymap

```
helloworld_Init(hw)
struct helloworld *hw;  {
    inset_InitStructure("helloworld", hw);
    hw->x = hw->y = helloworld_BADPOS;
    hw->newx = hw->newy = helloworld_BADPOS;
    hw->newblackonwhite = hw->blackonwhite = 1;
    hw->menus = im_NewML();
    im_AddToML(hw->menus, centerstring, "Center", hw, 0);
    im_AddToML(hw->menus, invertbackground, "Invert", hw, 0);

    hw->commands = keymap_create();
    keymap_insertproc(hw->commands, '\003', centerstring);
    keymap_insertproc(hw->commands, '\011', invertbackground);
    hw->state = keymap_newstate(hw->commands);
}
```

keymap_create () creates a new keymap by dynamically allocating memory for a keymap structure. In addition, it initializes the newly allocated keymap structure so that all the keymap entries have the status keymap_EMPTY, that is, no procedures and sub-keymaps are bound to characters. *keymap_create* returns a pointer to the newly created keymap, which we store in *hw->commands*.

keymap_insertproc (hw->commands, '\003', centerstring) sets the keymap *hw->commands* to return the procedure *centerstring* whenever the user types the character '\003,' a *ctrl-c*. *keymap_insertproc(hw->commands, '\011', invertbackground)* sets the keymap to the procedure *invertbackground*.

In addition to binding keys to procedures, it is possible to bind keys to other keymaps with the procedure *keymap_insertmap*. This is necessary for implementing sequences of keys, e.g., *ctrl-x ctrl-s*. See the section *The KeyMap Package* (p. 133 for a discussion.

hw->state =keymap_newstate (hw->commands) creates and initializes a new keystate for the keymap *hw->commands* and assigns it to *hw->state*, the keystate for the *helloworld* inset, *hw*.

If you are creating an inset named *x* and you want to use keymaps, then in the *x_Init* procedure for the inset, you should call *x_keymap_pointer = keymap_create ()* for each keymap that your inset needs. Then, you should call *keymap_insertproc* or *keymap_insertmap* as needed to bind procedures and sub-keymaps to character entries in the keymap. Finally, you should call *keymap_newstate* to create and initialize a keystate for the root keymap, and store the keystate with your inset.

Mapping sequences of keys

```
int helloworld_KeyIn(hw, c)
struct helloworld *hw;
int c;  {
    int result;

    result = keymap_char(hw->state, hw, c);
    if (result == inset_KEYACCEPTABLE) inset_WantUpdate(hw, hw);
    return result;
}
```

keymap_char (hw->state, hw, c) performs one step in mapping a sequence of characters to a procedure. It simulates the typing of the character *c* to the keystate *hw->state*. If *hw->state* maps the character to a procedure, *keymap_char* calls the procedure with two parameters, *hw* and *c*. (Although not done in this example, if *hw->state* were to map to a sub-keymap, *keymap_char* would set the *hw->state* so that the next character that the user typed would be mapped using the sub-keymap. See *The KeyMap Package*, p. 133, for an example).

keymap_char returns inset_KEYACCEPTABLE if procedure is called, inset_KEYUNACCEPTABLE if no binding for the character sequence was found, and inset_KEYPARTIALACCEPT if it is in the middle of processing a key sequence.

if (result = = inset_KEYACCEPTABLE) inset_WantUpdate(hw, hw) tests whether the result is inset_KEYACCEPTABLE and if it is, asks for an update.

In general, if you have defined a keymap for an inset *x*, then you should call *keymap_char* in your *x_KeyIn* procedure. Your *x_KeyIn* procedure should check the return value of *keymap_char*. If it is keymap_KEYACCEPTABLE, you should call *inset_WantUpdate* for the inset.

Static loading of the keymap

```
#include "CamphorImport.h"
#include "be2/inset.h"
#include "be2/im.h"
#include "be2/keymap.h"
#include "helloworld.h"

main()  {
    im_WindowPtr win;
    struct helloworld *hw;

    camphorinit(0, 0, 0, "/usr/andrew/lib/be2");
    staticload("im", imCamphorInitializer());
    staticload("inset", insetCamphorInitializer());
    staticload("keymap", keymapCamphorInitializer());
    staticload("helloworld",helloworldCamphorInitializer());
```

Since since we know in advance that it is needed, we will load the keymap package statically with the statement *staticload("keymap", keymapCamphorInitializer())* in the file *main.c*. Note that the keymap package must be included with the statement *#include "be2/keymap.h"* as well.

Program Listing for Example 6

helloworld.h

```
struct helloworld  {
    struct inset myinset; /* inset structure */
    int x,y; /* current location of the string */
    int newx, newy; /* new position for the string */
    char blackonwhite; /* true of text is to be drawn black on white */
    char newblackonwhite;
    struct menuList *menus; /* menus for this inset */
    struct keymap *commands;
    struct keystate *state;
};

BeginModule(helloworld, 6)
    Entry(helloworld_New, struct helloworld *)
    Entry(helloworld_Init, int)
    Entry(helloworld_FullUpdate, int)
    Entry(helloworld_Update, int)
    Entry(helloworld_Hit, struct inset *)
    Entry(helloworld_KeyIn, int)
EndModule()

#define helloworld_BADPOS 0x80000000
```

helloworld.c

```
#include "CamphorImport.h"
#include "be2/graphics.h"
#include "be2/inset.h"
#include "be2/im.h"
#include "be2/keymap.h"

#include "CamphorExport.h"
#include "helloworld.h"



centerstring(hw)
struct helloworld *hw;  {
    hw->newx =graphics_GetWidth / 2;
    hw->newy = graphics_GetHeight / 2;
    inset_WantUpdate(hw, hw);
}

invertbackground(hw)
struct helloworld *hw;  {
    hw->newblackonwhite = ! hw->newblackonwhite;
    inset_WantUpdate(hw, hw);
}
```

```
struct helloworld *helloworld_New()  {
    register struct helloworld *hw;

    hw = (struct helloworld *) malloc(sizeof(struct helloworld));

    helloworld_Init(hw);

    return(hw);
}

helloworld_Init(hw)
struct helloworld *hw;  {
    inset_InitStructure("helloworld", hw);
    hw->x = hw->y = helloworld_BADPOS;
    hw->newx = hw->newy = helloworld_BADPOS;
    hw->newblackonwhite = hw->blackonwhite = 1;
    hw->menus = im_NewML();
    im_AddToML(hw->menus, centerstring, "Center", hw, 0);
    im_AddToML(hw->menus, invertbackground, "Invert", hw, 0);
    hw->commands = keymap_create();
    keymap_insertproc(hw->commands, '\003', centerstring);
    keymap_insertproc(hw->commands, '\011', invertbackground);
    hw->state = keymap_newstate(hw->commands);
}

helloworld_FullUpdate(hw, how)
struct helloworld *hw;
int how;  {

    if (how == inset_FULLREDRAW || how == inset_LASTPARTIALREDRAW)  {
      if (how == inset_LASTPARTIALREDRAW)  {
          /* Have to clear out the rectangle since it is not cleared on a
      partial redraw */

          if (hw->blackonwhite)  {
            graphics_SetBackground(hw, graphics_WhitePixel)
          }
          else  {
            graphics_SetBackGround(hw, graphics_BlackPixel);
          }
          graphics_Clear(hw);
      }
      else if (! hw->blackonwhite) {
          /* Have to paint the rectangle black since it is set to white
      before a full redraw */

          graphics_SetBackGround(hw, BlackPixel);
          graphics_Clear(hw);
      }

      graphics_SetFunction(hw, graphics_GCinvert);
```

```
        if (hw->x == helloworld_BADPOS)  {
            hw->newx = hw->x = graphics_GetWidth(hw) / 2;
            hw->newy = hw->y = graphics_GetHeight (hw) / 2;
        }
        graphics_Text(hw, x, y, "hello world");

        inset_WantMenuList(hw, hw, hw->menus);
    }
}

helloworld_Update(hw)
struct helloworld *hw;  {

    if (hw->newblackonwhite != hw->blackonwhite)  {
        graphics_SetFunction(hw, graphics_GCinvert);
        graphics_FillRectangle(hw, graphics_GetWidth(hw),
        graphics_GetHeight(hw));
        hw->blackonwhite = hw->newblackonwhite;
    }
    if (hw->newx != hw->x || hw->newy != hw->y)  {
        graphics_SetFunction(hw, graphics_GCinvert);
        graphics_Text(hw, x, y, "hello world");
        hw->x = hw->newx;
        hw->y = hw->newy;
        graphics_Text(hw, x, y, "hello world");
        graphics_SetFunction(hw, graphics_GCset);
    }
}

struct inset *helloworld_Hit(hw, action, x, y)
struct helloworld *hw;
int action, x, y;  {
    if (action == MouseMask(LeftButton, DownTransition) || action ==
      MouseMask(RightButton, DownTransition))  {
        hw->newx = x;
        hw->newy = y;
        inset_WantUpdate(hw, hw);
    }
    return (struct inset *) hw;
}

int helloworld_KeyIn(hw, c)
struct helloworld *hw;
int c;  {
    int result;

    result = keymap_char(hw->state, hw, c);
    if (result == inset_KEYACCEPTABLE) inset_WantUpdate(hw, hw);
    return result;
}
```

main.c

```
#include "CamphorImport.h"
#include "be2/inset.h"
#include "be2/im.h"
#include "be2/keymap.h"
#include "helloworld.h"

main()  {
    im_WindowPtr win;
    struct helloworld *hw;

    camphorinit(0, 0, 0, "/usr/andrew/lib/be2");
    staticload("im", imCamphorInitializer());
    staticload("inset", insetCamphorInitializer());
    staticload("keymap", keymapCamphorInitializer());
    staticload("helloworld",helloworldCamphorInitializer());

    hw = (struct helloworld *) inset_New("helloworld", 0);
    win = im_CreateWindow(0);
    im_FillWindow(win, hw);
    inset_WantInputFocus(hw, hw);

    im_KeyboardProcessor();
}
```

Makefile

```
CFLAGS = -g -DSTATICLINKING -I"/usr/andrew/include"

.SUFFIXES:  .o

helloworld.o: helloworld.c helloworld.h

helloworld: main.o helloworld.o
      cc  -g -o hw hwmain.o helloworld.o  /usr/andrew/lib/libbx.a
      /usr/andrew/lib/libitc.a
```

Example 7: Working with scroll bars

Example 6 showed how to use the keymap package in responding to keyboard input. This section will show how to add scroll bars to insets. This particular example will have both vertical and horizontal scroll bars.

Running the example program

Before reading the discussion of the example program, you may find it helpful to run the program on your workstation.

Action 1. To run the program on your workstation, at the **Typescript** prompt, type

```
/usr/andrew/doc/example7/helloworld
```

and press the Enter key.

Response. The program will produce a window with *hello world* centered in the inset's rectangle; in this case, in the body of the window..

Action 2. To move *hello world* in the window, position the mouse cursor within the program's window and click on the left mouse button.

Response. *hello world* will be drawn, centered on the point where you clicked the left mouse button.

Action 3. To use the keyboard to center *hello world* in the center of the inset's rectangle, type

```
ctrl-c
```

Response. *hello world* will be drawn in the center of the inset's rectangle.

Action 4. To invert the drawing, type

```
ctrl-i
```

Response. The background will invert. In this case, it will change from white to black. Action 3 and 4 can be repeated in any order.

Action 5. Use the vertical scroll bar to scroll up or down.

Response. The inset will scroll up or down.

Action 6. Use the horizontal scroll bar to scroll to the left or right.

Response. The inset will scroll to the left or the right..

Action 7. To quit the program, move your mouse cursor into the window's title bar, pop up the menus and choose **Zap** from the *This Window* menu card.

Response. The window will disappear from the screen.

Creating the inset

Naming the inset

For this example, we will continue to use the inset name, *helloworld*.

Declaring the data structure for the inset

```
struct helloworld  {
    struct inset myinset; /* inset structure */
    int x,y; /* current location of the string */
    int newx, newy; /* new position for the string */
    char blackonwhite; /* true if text is to be drawn black on
      white */
    char newblackonwhite;
    struct menuList *menus; /* menus for this inset */
    struct keymap *commands;
    struct keystate *state;
};
```

The data structure for this example is the same as in Example 6.

Defining the driver routines for the inset

Example 7 requires six more driver routines than Example 6. The additional routines are: a routine to get information from the inset when using a vertical scroll bar, a routine to find something in a given y pixel location, a routine to setup the display with respect to a vertical scroll bar, a routine to get information when using horizontal scroll bars, a routine to find something in a given x pixel location, and a routine to setup the display with respect to a horizontal scroll bar. These six routines are the minimum necessary to set up both vertical and horizontal scroll bars for an inset.

Procedures for centering a string and inverting the background

```
centerstring(hw)
struct helloworld *hw;  {
    hw->newx = hw->graphics_GetWidth(hw) / 2;
    hw->newy = hw->graphics_GetHeight(hw) / 2;
    inset_WantUpdate(hw, hw);
}

invertbackground(hw)
struct helloworld *hw;  {
    hw->newblackonwhite = ! hw->newblackonwhite;
    inset_WantUpdate(hw, hw);
}
```

These procedures are the same as in previous examples. See Example 3 for a description.

Similarly, the fullupdate, update, menu, and keyIn procedures are the same as in the previous example. For a listing, see the end of this section. The procedures are described in more detail in Example 6.

Setting up the scroll bars

Each scroll bar needs a minimum of three routines, that is, three routines for the vertical scroll bar, and three for the horizontal scroll bar. These routines provide information about the inset to the scroll bar module, so the scroll bars will know how to properly draw themselves in the inset window, and so they can tell the inset how to redraw when there is a movement on the scroll bars.

Getting information for the vertical scroll bar

```
helloworld_yGetInfo(hw, len, vstart, vsize, dstart, dsize)
struct helloworld *hw;
int *len, *vstart, *vsize, *dstart, *dsize;  {
    *len = graphics_GetHeight(hw);
    *vstart = 0;
    *dstart = hw->y - graphics_GetVisualTop(hw);
    if (hw->y < graphics_GetVisualTop(hw))  {
       *len -= hw->y;
       *vstart -= hw->y;
       *dstart -= hw->y;
    }
    else if (hw->y > graphics_GetVisualTop(hw) +
       graphics_GetHeight(hw))   {
       *len = hw->y;
    }
    *vsize = graphics_GetHeight(hw);
    *dsize = 1;
}
```

```
helloworld_yWhatIsAt(hw, loc, pos)
struct helloworld *hw;
int loc;
int *pos;  {
    if (hw->y < graphics_GetVisualTop(hw))  {
       *pos = hw->y + loc;
    }
    else  {
       *pos = loc;
    }
}

helloworld_ySetFrame(hw, pos, ylocn)
struct helloworld *hw;
int pos, ylocn;  {
    hw->newy = hw->y + ylocn - pos;
    inset_WantUpdate(hw,hw);
}
```

The procedures *yGetInfo*, *yWhatIsAt*, and *ySetFrame* provide the vertical scroll bar with the necessary information to draw or redraw itself in the inset, and in turn, the scroll bar will tell the inset how to redraw itself according to movements on the scroll bar.

yGetInfo is called to discover information about the inset. The inset should return its length in *len*, what part of the object is visible in *vstart* and *vsize*, and what part of the object is selected in *dstart* and *dsize*.

yWhatIsAt returns in *pos* the position in the inset corresponding to *loc* in pixels from the top of the inset.

ySetFrame tells the inset to setup its display so that the data at position *pos* appears at *where* pixels from the top of the inset. A view moves the line that contains *pos* to be close to *where* pixels away from the top of the rectangle.

Setting up the horizontal scroll bar

```
helloworld_xGetInfo(hw, len, vstart, vsize, dstart, dsize)
struct helloworld *hw;
int *len, *vstart, *vsize, *dstart, *dsize;  {
    *len = graphics_GetWidth(hw);
    *vstart = 0;
    *dstart = hw->x - graphics_GetVisualLeft(hw);
    if (hw->x < graphics_GetVisualLeft(hw))  {
       *len -= hw->x;
       *vstart -= hw->x;
       *dstart -= hw->x;
    }
    else if (hw->x > graphics_GetVisualLeft(hw) +
       graphics_GetWidth(hw))   {
       *len = hw->x;
```

```
    }
    *vsize = graphics_GetWidth(hw);
    *dsize = 1;
}

helloworld_xWhatIsAt(hw, loc, pos)
struct helloworld *hw;
int loc;
int *pos;  {
    if (hw->x < graphics_GetVisualLeft(hw))  {
      *pos = hw->x + loc;
    }
    else  {
      *pos = loc;
    }
}

helloworld_xSetFrame(hw, pos, xlocn)
struct helloworld *hw;
int pos, xlocn;  {
    hw->newx = hw->x + xlocn - pos;
    inset_WantUpdate(hw, hw);
}
```

The procedures *xGetInfo*, *xWhatIsAt*, and *xSetFrame* are exactly like their vertical scroll bar counterparts.

Exporting and importing routines

Declaring the driver routines for export

In this example, *helloworld.h* will contain the following declaration:

```
BeginModule(helloworld, 12)
    Entry(helloworld_New, struct helloworld *)
    Entry(helloworld_Init, int)
    Entry(helloworld_FullUpdate, int)
    Entry(helloworld_Update, int)
    Entry(helloworld_Hit, struct inset *)
    Entry(helloworld_KeyIn, int)
    Entry(helloworld_yGetInfo, int)
    Entry(helloworld_yWhatIsAt, int)
    Entry(helloworld_ySetFrame, int)
    Entry(helloworld_xGetInfo, int)
    Entry(helloworld_xWhatIsAt, int)
    Entry(helloworld_xSetFrame, int)
EndModule()

#define helloworld_BADPOS 0x80000000
```

Importing BE2 routines and exporting driver routines

In this example, the *helloworld.c* file should begin with the following declarations:

```
#include "CamphorImport.h"
#include "be2/graphics.h"
#include "be2/inset.h"
#include "be2/im.h"
#include "be2/keymap.h"
#include "be2/scroll.h"

#include "CamphorExport.h"
#include "helloworld.h"
```

Note the addition of "be2/scroll.h" module.

Similarly, for the file *main.c*, the declarations should be:

```
#include "CamphorImport.h"
#include "be2/inset.h"
#include "be2/im.h"
#include "be2/keymap.h"
#include "be2/scroll.h"
#include "helloworld.h"
```

Setting up the inset

```
main()  {
    im_WindowPtr win;
    struct helloworld *hw;
    struct inset *sb;

    camphorinit(0, 0, 0, "/usr/andrew/lib/be2");
    staticload("im", imCamphorInitializer());
    staticload("inset", insetCamphorInitializer());
    staticload("keymap", keymapCamphorInitializer());
    staticload("scroll", scrollCamphorInitializer());
    staticload("helloworld",helloworldCamphorInitializer());

    im_init();
    hw = (struct helloworld *) inset_New("helloworld", 0);
    sb = inset_New("scroll", 0, hw);
    scroll_SetStates(sb, scroll_LEFT, scroll_TOP);
    win = im_CreateWindow(0);
    im_FillWindow(win, sb);
    inset_WantInputFocus(hw, hw);

    im_KeyboardProcessor();
}
```

The main.c file is similar to that for Example 6 with the additional static loading of scroll bar package and the setting of scroll bar inset.

Compiling the inset

```
CFLAGS = -g -DSTATICLINKING -I"/usr/andrew/include"

.SUFFIXES:  .o

helloworld.o: helloworld.c helloworld.h

helloworld: main.o helloworld.o
        cc  -g -o hw main.o helloworld.o
        /usr/andrew/lib/libbe2.a\
                /usr/andrew/lib/libitc.a
```

The Makefile for this example is exactly as in the previous examples, and is a static link compilation.

Program Listing for Example 7

helloworld.h

```
struct helloworld  {
    struct inset myinset; /* inset structure */
    int x,y; /* current location of the string */
    int newx, newy; /* new position for the string */
    char blackonwhite; /* true of text is to be drawn black on white */
    char newblackonwhite;
    struct menuList *menus; /* menus for this inset */
    struct keymap *commands;
    struct keystate *state;
};

BeginModule(helloworld, 12)
    Entry(helloworld_New, struct helloworld *)
    Entry(helloworld_Init, int)
    Entry(helloworld_FullUpdate, int)
    Entry(helloworld_Update, int)
    Entry(helloworld_Hit, struct inset *)
    Entry(helloworld_KeyIn, int)
    Entry(helloworld_yGetInfo, int)
    Entry(helloworld_yWhatIsAt, int)
    Entry(helloworld_ySetFrame, int)
    Entry(helloworld_xGetInfo, int)
    Entry(helloworld_xWhatIsAt, int)
    Entry(helloworld_xSetFrame, int)
EndModule()

#define helloworld_BADPOS 0x80000000
```

helloworld.c

```
#include "CamphorImport.h"
#include "be2/graphics.h"
#include "be2/inset.h"
#include "be2/im.h"
#include "be2/keymap.h"
#include "be2/scroll.h"

#include "CamphorExport.h"
#include "helloworld.h"

centerstring(hw)
struct helloworld *hw;  {
    hw->newx = hw->graphics_GetWidth(hw) / 2;
    hw->newy = hw->graphics_GetHeight(hw) / 2;
    inset_WantUpdate(hw, hw);
}
```

```
invertbackground(hw)
struct helloworld *hw;  {
    hw->newblackonwhite = ! hw->newblackonwhite;
    inset_WantUpdate(hw, hw);
}

struct helloworld *helloworld_New()  {
    register struct helloworld *hw;

    hw = (struct helloworld *) malloc(sizeof(struct helloworld));

    helloworld_Init(hw);

    return(hw);
}

helloworld_Init(hw)
struct helloworld *hw;  {
    inset_InitStructure("helloworld", hw);
    hw->x = hw->y = helloworld_BADPOS;
    hw->newx = hw->newy = helloworld_BADPOS;
    hw->newblackonwhite = hw->blackonwhite = 1;
    hw->menus = im_NewML();
    im_AddToML(hw->menus, centerstring, "Center", hw, 0);
    im_AddToML(hw->menus, invertbackground, "Invert", hw, 0);
    hw->commands = keymap_create();
    keymap_insertproc(hw->commands, '\003', centerstring);
    keymap_insertproc(hw->commands, '\011', invertbackground);
    hw->state = keymap_newstate(hw->commands);
}

helloworld_FullUpdate(hw, how)
struct helloworld *hw;
int how;  {

    if (how == inset_FULLREDRAW || how == inset_LASTPARTIALREDRAW)  {
      if (how == inset_LASTPARTIALREDRAW)  {
          /* Have to clear out the rectangle since it is not cleared on a
      partial redraw */

          if (hw->blackonwhite)  {
            graphics_SetBackground(hw, graphics_WhitePixel)
          }
          else  {
            graphics_SetBackGround(hw, graphics_BlackPixel);
          }
          graphics_Clear(hw);
      }
      else if (! hw->blackonwhite) {
          /* Have to paint the rectangle black since it is set to white
      before a full redraw */
```

```
            graphics_SetBackGround(hw, BlackPixel);
            graphics_Clear(hw);
        }

        graphics_SetFunction(hw, graphics_GCinvert);

        if (hw->x == helloworld_BADPOS)  {
            hw->newx = hw->x = graphics_GetVisualLeft(hw) +
        graphics_GetWidth(hw) / 2;
            hw->newy = hw->y = graphics_GetVisualTop(hw) +
        graphics_GetHeight(hw) / 2;
        }

        graphics_Text(hw, x, y, "hello world");

        inset_WantMenuList(hw, hw, hw->menus);
    }
}

helloworld_Update(hw)
struct helloworld *hw;  {

    if (hw->newblackonwhite != hw->blackonwhite)  {
        graphics_SetFunction(hw, graphics_GCinvert);
        graphics_FillRectangle(hw, graphics_GetVisualLeft(hw),
        graphics_GetVisualTop(hw), graphics_GetWidth(hw),
        graphics_GetHeight(hw));
        hw->blackonwhite = hw->newblackonwhite;
    }
    if (hw->newx != hw->x || hw->newy != hw->y)  {
        graphics_SetFunction(hw, graphics_GCinvert);
        graphics_Text(hw, x, y, "hello world");
        hw->x = hw->newx;
        hw->y = hw->newy;
        graphics_Text(hw, x, y, "hello world");
        graphics_SetFunction(hw, graphics_GCset);
    }
}

struct inset *helloworld_Hit(hw, action, x, y)
struct helloworld *hw;
int action, x, y;  {
    if (action == MouseMask(LeftButton, DownTransition) || action ==
      MouseMask(RightButton, DownTransition))  {
        hw->newx = x;
        hw->newy = y;
        inset_WantUpdate(hw, hw);
    }

    return (struct inset *) hw;
```

```
}

int helloworld_KeyIn(hw, c)
struct helloworld *hw;
int c;  {
    int result;

    result = keymap_char(hw->state, hw, c);
    if (result == 0) inset_WantUpdate(hw, hw);
    return result;
}

helloworld_yGetInfo(hw, len, vstart, vsize, dstart, dsize)
struct helloworld *hw;
int *len, *vstart, *vsize, *dstart, *dsize;  {
    *len = graphics_GetHeight(hw);
    *vstart = 0;
    *dstart = hw->y - graphics_GetVisualTop(hw);
    if (hw->y < graphics_GetVisualTop(hw))  {
       *len -= hw->y;
       *vstart -= hw->y;
       *dstart -= hw->y;
    }
    else if (hw->y > graphics_GetVisualTop(hw) + graphics_GetHeight(hw))
       {
       *len = hw->y;
    }
    *vsize = graphics_GetHeight(hw);
    *dsize = 1;
}

helloworld_yWhatIsAt(hw, loc, pos)
struct helloworld *hw;
int loc;
int *pos;  {
    if (hw->y < graphics_GetVisualTop(hw))  {
       *pos = hw->y + loc;
    }
    else  {
       *pos = loc;
    }
}

helloworld_ySetFrame(hw, pos, ylocn)
struct helloworld *hw;
int pos, ylocn;  {
    hw->newy = hw->y + ylocn - pos;
    inset_WantUpdate(hw,hw);
}

helloworld_xGetInfo(hw, len, vstart, vsize, dstart, dsize)
```

```
struct helloworld *hw;
int *len, *vstart, *vsize, *dstart, *dsize;  {
    *len = graphics_GetWidth(hw);
    *vstart = 0;
    *dstart = hw->x - graphics_GetVisualLeft(hw);
    if (hw->x < graphics_GetVisualLeft(hw))  {
       *len -= hw->x;
       *vstart -= hw->x;
       *dstart -= hw->x;
    }
    else if (hw->x > graphics_GetVisualLeft(hw) + graphics_GetWidth(hw))
       {
       *len = hw->x;
    }
    *vsize = graphics_GetWidth(hw);
    *dsize = 1;
}

helloworld_xWhatIsAt(hw, loc, pos)
struct helloworld *hw;
int loc;
int *pos;  {
    if (hw->x < graphics_GetVisualLeft(hw))  {
       *pos = hw->x + loc;
    }
    else  {
       *pos = loc;
    }
}

helloworld_xSetFrame(hw, pos, xlocn)
struct helloworld *hw;
int pos, xlocn;  {
    hw->newx = hw->x + xlocn - pos;
    inset_WantUpdate(hw, hw);
}
```

main.c

```
#include "CamphorImport.h"
#include "be2/inset.h"
#include "be2/im.h"
#include "be2/keymap.h"
#include "be2/scroll.h"
#include "helloworld.h"

main()  {
    im_WindowPtr win;
    struct helloworld *hw;
    struct inset *sb;
```

```
    camphorinit(0, 0, 0, "/usr/andrew/lib/be2");
    staticload("im", imCamphorInitializer());
    staticload("inset", insetCamphorInitializer());
    staticload("keymap", keymapCamphorInitializer());
    staticload("scroll", scrollCamphorInitializer());
    staticload("helloworld",helloworldCamphorInitializer());

    im_init();
    hw = (struct helloworld *) inset_New("helloworld", 0);
    sb = inset_New("scroll", 0, hw);
    scroll_SetStates(sb, scroll_LEFT, scroll_TOP);
    win = im_CreateWindow(0);
    im_FillWindow(win, sb);
    inset_WantInputFocus(hw, hw);

    im_KeyboardProcessor();
}
```

Makefile

```
CFLAGS = -g -DSTATICLINKING -I"/usr/andrew/include"

.SUFFIXES:  .o

helloworld.o: helloworld.c helloworld.h

helloworld: main.o helloworld.o
      cc  -g -o hw main.o helloworld.o  /usr/andrew/lib/libbe2.a\
            /usr/andrew/lib/libitc.a
```

Programming Environment

In addition to setting up the proper inset (and/or data object) structure, it is necessary to compile the program within the proper environment. The programmer's interface to both insets and data objects in BE2 requires certain operations on procedures that are normally not provided in the C language.

In particular, BE2 must periodically, given the name of a type of inset or data object, generate a set of procedures that implement specific operations on objects of that type. BE2 does this by combining the name of the type of object and the name of the operation into a character string which calls the procedure implementing the operations neccessary on objects of that type.

Thus, BE2 requires an operation that efficiently maps character strings into procedure pointers. This section describes this operation, and related mechanisms, which are provided by the *Camphor* programming environment.

Procedure names in the Camphor environment are composed of a module name and an entry name within that module. The procedure name is represented as a character string, with an underscore separating the module name from the entry name. For example, the procedure named *Redraw* within the *piechart module* would be named as *piechart_Redraw*. In most cases, all of the procedures in a single module are expected to exist in a single source file, and some macros that simplify operations in this case have been provided.

One of the major advantages of Camphor is that is supports dynamic loading of procedures. Therefore, it is possible to have, say, a picture inset dynamically loaded into a text document, which saves a lot of time. In the previous chapter, the examples dealt only with static linking. This chapter will deal more with dynamic loading procedures, but this does not at all mean that you will always need to dynamically load in BE2. In particular, the Import and Export procedures described in the next section is important whether you are using static or dynamic load.

Imports and Exports

For every code module provided, there must be an include file enumerating the procedures exported by that module. This should greatly simplify the usage dynamic linking procedures in Camphor. The procedures must be declared by means of a macro, *Entry*, which gives both the type and name of each procedure. This include file, depending upon the definition of the Entry macro, can be used to import functions from a module or to export functions from a module.

A short example should clarify things:

```
#include "CamphorImport.h"
#include "importedModule1.h" /* import functions from module */
#include "importedModule2.h" /* and this one, */
#include "importedModule3.h" /* and this one, */
```

```
#include "importedModule4.h" /* and this one. */

#include "CamphorExport.h"
#include "myModule.h" /* these functions are exported */
```

The include files CamphorImport.h and CamphorExport.h both define their own versions of the BeginModule, Entry, and EndModule macros. The former defines these macros to generate the code required to import procedures from another module within the Camphor run-time system, while the latter defines these macros to generate the code required to export procedures from a module.

BeginModule defines the start of the module definition. It gives the module name and, unfortunately, a number which must be at least as large as the number of entries listed following. (Unfortunate because this number must be changed every time the entries are changed) After the BeginModule line, there are a number of lines giving the name of the exported procedure, and the type of value that it returns. Finally, the module definition completes with an EndModule line. See any of the *main.c* files for the examples in the previous section.

Dynamic Loading

When a reference to a module is made (either by calling an imported procedure, or by calling type_makeproc or type_create procedures described below) and the required module is not present in the running program, the Camphor run-time system can attempt to search for and load the missing module at run-time. Essentially,a path to be searched can be set up before the reference to the procedure,and when the reference is made, the dynamic loader will search the path for the module, load it if found, and augment the symbol table enumerating the modules and procedures whose addresses are known.

This dynamically loading of modules makes it convenient for people to write their own inset code independent of the maintainers of the library and other applications making use of the library, while still allowing these new, user-defined insets to be used.

With static linking, the linker combines several object programs into a single program, searching libraries and resolving all external references. If there are no errors, the output of the linker is an executable file (see *ld (1)* in the Unix man or Andrew on-line help pages).

With dynamic linking, all procedure declarations are transformed to be illegal instruction traps.
These are then handled by a signal handler.

If STATICLINKING is defined, the procedure declarations are converted to extern type proc (). This is more efficient if you do not need the inset to be dynamically loaded.

If the inset you are defining is not going to be dynamically loaded by a multi-media editor, you should define STATICLINKING since it allows the preprocessor to generate more efficient code. On the other hand, if you are writing an inset for general use (e.g., scroll bars, drawing editor insets, etc.), you should remove the -DSTATICLINKING option from your compilation flags.

-Dname = def Defines the name to the preprocessor, as if by #define. If no definition is give, the name is defined as 1.

If the inset you are defining is going to be dynamically loaded, you must include the following in your Makefile:

```
.SUFFIXES:  .6.o

o.6:

   make6 $@
```

make6 is a shell script (located in /user/andrew/bin) that runs the linker to produce a relocatable object module. Relocatable object modules have their references to relative positions rather than absolute addresses. Relocatable modules are necessary in order for dynamic loading to work. Relative addresses are usually less efficient than absolute addresses, so if you are not going to dynamically load, you should remove this line from the Makefile.

Debugging

Check that each routine for the inset driver in <inset-name>.c is exported in <inset-name>.h

Check that the number of exports in BeginModule (n) matches or exceeds the number (less efficient) the number of Entry's in the list.

Dynamic Procedure Linking --- The Type Module Calls

Support for run-time linking of procedures is provided by the *type module* procedure. Type module provides two routines, *type_create* and *type_makeproc*, for manipulating collections of procedures by name.

The routine *type_makeproc* maps a procedure name to a pointer to the actual code. It takes two parameters, a module name and an entry name, and returns a pointer to the appropriate procedure, or null if the procedure cannot be located.

For example, the module myModule.h from the first example about might look like this:

```
BeginModule(myModule, 3)
 Entry(myModule_Proc1, int)
 Entry(myModule_Proc2, char *)
 Entry(myModule_Proc3, float)
```

```
EndModule()
```

The routine *type_create*, on the other hand, deals with entire modules. It takes four parameters: an interface name, a module name, an array of procedure names within that module that make up the interface, and the address of a structure in which to store the located procedures. Essentially, this procedure provides a bulk interface to the dynamic type routines, so that instantiating many objects of the same type does not result in repeatedly looking up all the names.

An interface consists of an ordered set of procedures, and several different modules can export the same interface. In addition, any one module can also export several different interfaces. When *type_create* is called, it first checks its list of located interfaces to see if it has located the procedures defining the named interface for the particular module also named in the call. If it has, *type_create* simply copies the list of procedure addresses from its table to the structure passed by the user to contain these addresses. It is only in the case where *type_create* has not yet instantiated the interface for the requested module that it makes use its third parameter: the array of procedure names making up the interface. In this case, *type_create* calls *type_makeproc* repeatedly with the name of the module it is instantiating and each procedure from the array of procedure names.

Data Stream

This section describes the data stream protocol definition, i.e., the specifications for data objects' external data representation. The discussion centers around a data stream for a text data object. Other, more global, issues are touched upon, but only as they refer to text data objects.

The external data representation is expected to be used both for storing objects in files and for cut and paste operations.

Goals for a data stream protocol

-- Handle an arbitrary collection of insets and data objects. Achieved through the use of \begindata, \enddata and \inset commands.

-- Share data objects between two views in the same document. Achieved by the separation of the \begindata and \enddata commands from the \inset command and the inclusion of a data object registry.

-- Specify an inset (identified by name) so that commands can be sent to it. Achieved by the use of an inset registry.

-- Contain only printable ASCII characters (make the datastream more readable). Achieved through expanded versions of stylesheet definitions and command names.

-- Simple parsing algortihm (easy to determine beginning and end of data objects). Achieved through use of simplified command names and recursive inset parsing.

-- Interpreted and uninterpreted files should be recognized by the system and treated accordingly. (reserved characters should not have to be quoted in uninterpreted files). Achieved by checking for the version string \dsversion{#} at the beginning of the highest level data object.

Protocol Definition

Reserved Characters

\
{
}

These characters are reserved, and should be quoted with a \ (backslash) in interpreted documents. Quoting should not be used in uninterpreted documents.

Version Header

\dsversion{*versionnumber*}

This string should be included as the first line of the highest level data object in the document. This string is used to distinguish formatted documents from plain text documents. *Versionnumber* refers to the version of the new datastream that was used to write the document and it determines which read routine will be used to read it.

Stylesheet Definitions

\define{*stylesheetname*, *devicetype*, *menuname*
[*attribute opcode optype opparm*]
.
.
[*attribute opcode optype opparm*]}

A stylesheet is a series of attribute modifications which create the desired effect on the selected text. The stylesheet format above will be used both for stylesheet definitons in the document and for those in the template. *Stylesheetname* refers to the common name of the stylesheet (e.g. italic, bold, etc). *Devicetype* refers to the display medium (e.g. D for display, P for printer). *Menuname* is the internal name of the stylesheet (e.g. Font,Italic, Font,Bold, etc). A stylesheet can contain one or more attribute specifications. *Attribute* refers to the name of the attribute being modified (e.g. fontface, leftmargin). *Opcode* is the operator to be used in modifying that attribute (e.g. copy, add). *Optype* specifies the type of the parameter (e.g. points, inches). *Opparm* is a parameter of the operation which can be either a digit or a string (e.g. "AndyType", 42).

Template Inclusion

\template{*templatename*}

The template string may be used in a text document to indicate that a template should be read before the document is processed. *Templatename* refers to the name of the template to be used. *Templatename* may include a full path (e.g. /cmu/itc/maria/templates/my.template), or may use the default path (/usr/andrew/lib/Templates/xxx.template, where the user has typed in the 'xxx' portion of the path) or a path that is defined in the user's preference file. The inclusion of a template string in the file does not preclude the inclusion of additional stylesheet definitions, however a document should include only one template string.

Insets and Data Objects

\begindata{*datatype*, *dataname*}
text
\enddata{*datatype*}

The begindata, enddata pair mark the extent of the data included in a particular data object. Use of the begindata, enddata pair indicate that the data should be stored in memory, and the location of the data and it's *dataname*, a canonical name used to refer to that data object, should be stored and registered in a data registry. *Datatype* refers to the type of data being stored (e.g. text, array, etc).

\inset{insettype, insetname, dataname}

The inset string is used to mark the place in the document where the inset should appear. The definition of the data object (it's begindata, enddata markers) and the inset marker need not be adjacent, however a data object definition must precede any inset marker which refers to it. Multiple inset markers may refer to a single data object. *Insettype* refers to the body of code which can be dynamically loaded to be used in displaying the inset (e.g. delta, eq, zip, etc.). *Insetname* is a canonical name used to refer to the inset and is registered in an inset registry. The *insetname* is also used to register the inset as a viewer of a particular data object. *Dataname* is a reference to the data object being used by this inset.

Styles in the data

stylesheetname{*text ... more text*}

A backslash followed by a command which is not one of the above commands, will be interpreted as a stylesheet command. That is, the indicated stylesheet should be used to modify the selected text. *Stylesheetname* refers to the name of the stylesheet to be used in the modification of the document. An error will result if a stylesheet with that name is not found. The curly brackets define the extent of the text for which the appropriate stylesheet modifications will be in effect.

Routines, Objects, and Support Packages

This section describes all the routines, useful objects, and assorted support packages that have been written for BE2. The sections in this chapter are as follows:

1) Inset Routines
2) Inset/Window Management Routines
3) Data Object Routines
4) Menu Routines
5) KeyMap Package
6) Document Objects
7) Dialog Box Inset
8) Scroll Bar Inset
9) Layout Pair Inset
10) Buffer Package
11) Update List Package
12) Key Recording Package

Inset Routines

This section discusses two groups of routines: (1) *x_* routines that you must provide when you define an inset named *x* and (2) *inset_* routines, located in *inset.c*, that you must use to call upon a particular inset's *x_* routines. The discussion divides the routines into the following groups:

Creating, initializing, and deleting insets
Negotiating about the size of the inset in the window
Adding menus for the inset
Managing mouse hits
Receiving and losing the input focus
Working with keyboard input
Redrawing and update requests

For each *x_* routine, there is a corresponding *inset_* routine or macro. The *inset_* routine typically sets up the inset so that its information is current, and then calls the corresponding *x_* routine. You should always use the *inset_* routines to call the *x_* routines for two reasons: First, it is easier for you to call an an *inset_* routine than to call the *x_* routine, because you don't need to manage the set up. Second, it is easier to define an inset *x* if it can assume that certain conditions will be true when it is called.

The structure of an inset

When you define an inset *x*, you must include an *inset* structure, defined in *inset.h*. The following describes the current *inset* structure as it appears in *inset.h*.

```
struct inset {
    char *insetname;            /* x, name of the inset named x */
    struct insetroutines *ir;   /* the inset routines */
    struct GraphicsState gs;    /* the graphics state associated with the inset */
    struct inset *parent; /* parent inset */
    char changed;       /* true is Update will be called */
    char flags;                 /* some flags for top level inset */
};
```

Rectangles and visible rectangles

Each inset has two rectangles associated with it: *r*, the rectangle that gives the dimensions that the inset can scale its drawing to, and *vr*, the visible rectangle that provides the dimensions for what is visible in the window. To understand *r* and *vr*, suppose that a multi-media editor were displaying some text and a drawing, and that the drawing happens to be at the very bottom of the window. If the drawing inset's rectangle, *r*, falls outside the window boundary, and the drawing is relative to *r*, then the drawing will be clipped to the inset's visible rectangle, *vr*, as shown in the figure below.

[To see a picture, zip /cmu /unix /itcsrc /itc /insets /documentation /be2docs /Inset.vr.zip]

You must attend to the visible rectangle if you wish to:

-- Scale the size of your drawing to the size of its visible region. In this case, you would do your drawing relative to the visible rectangle rather than the rectangle.

-- Produce a clipped drawing with maximum efficiency. In this case, you would scale your drawing relative to *r*, but you would not actually draw except when the coordinates fell within *vr*.

-- Save the visible region or a part of it in a buffer, via *im_SaveRegion*. In this case, the rectangle coordinates you pass to *im_SaveRegion* must be less than or equal to the rectangle, *vr;* otherwise, the results will be undefined.

-- Work with child insets. In this case, whenever you set the value of a child inset's rectangle, r, you must also set the value of its visible rectangle, vr. See *inset_FullUpdate* and *inset_ClipVisualRectangle* for discussions of how to work with child insets.

Otherwise you can ignore *vr* and simply draw within the dimensions given by *r*. If you do, your drawing will automatically be clipped to *vr*.

Creating and deleting insets

Providing a creation routine for inset x

```
struct x *x_New(dop)
struct x_data dop;
```

Description. If you are defining an inset named *x*, you must provide the routine *x_New*. You must declare *x_New* with a single parameter, *dop*, a pointer to the data object for the inset. *x_New* should dynamically allocate inset *x* (see *malloc (3)*) and then call *x_Init*.

Return value. If successful, *x_New* should return a pointer to the struct *x* that it has created; NULL for failure.

Usage. You should refer to *BE2 Basics*, pp. 11 - 79 for examples of how to work with the *x_New* construct.

Example. If you were defining an inset named *doc* with a *text* data object, you would have to create a procedure called *doc_New (text_pointer)*.

Creating an inset

```
struct inset *inset_New(name, parent, dataobject)
char *name;
struct inset parent;
long dataobject;
```

Description. *inset_New* creates and initializes an instance of the inset *name*. The parameter *parent* should be a pointer to the newly created inset's parent, or NULL if the newly created inset has no parent. *dataobject* should be a pointer to the data object that the newly created inset will display.

To create and initialize most of the inset structure, *inset_New* calls the corresponding *name*_New procedure for the inset *name*; then it initializes the data portion of the inset structure.

Return value. If successful, *inset_New* returns the pointer to the newly created inset; if the inset given by *name* does not exist or the inset could not be created for some other reason, then *inset_New* returns NULL.

Usage. You should always use the *inset_New* procedure to create an inset of type *name*, rather than calling *name*_New directly.

Example. *inset_New ("doc", 0, text)* creates and initializes a new document inset.

Providing an initialization routine for inset x

```
x_Init(ip, dop)
struct x_inset ip;
struct x_data dop;
```

Description. If you are defining an inset named *x*, you must provide the routine *x_Init*. You must declare *x_Init* with two parameters: *ip*, a pointer to the inset *x*, and *dop*, a pointer to the data object for the inset. *x_Init* should initialize any parts of the inset *x* structure that are local, set *x_inset->d* to *dop*, and initialize any data as necessary.

Usage. You should refer to *BE2 Basics*, pp. 11 - 79 for examples of how to work with the *x_Init* construct.

Example. If you were defining an inset named *doc* with a *text* data object, you would have to create a procedure called *doc_Init (doc_pointer, text_pointer)*.

Initializing an inset

```
inset_Init(name, ip, dop)
char name;
long ip, dop;
```

Description. *inset_Init* calls the *name_Init* routine for the inset given by *name*, passing it *ip*, a pointer to the inset given of type name, and *dop*, the data object associated with *ip*, as parameters.

Usage. *inset_Init* is not implemented (9/12/86). At the present time, you should call the *name_inset* routine directly in order to initialize it.

Providing a deletion routine for inset x

```
x_Destroy(ip)
struct inset ip;
```

Description. If you are defining an inset named *x*, you may optionally provide the routine *x_Destroy*. If you do define it, you must declare *x_Destroy* with a single parameter, *ip*, a pointer to the data object for the inset. *x_Destroy* should call *free(ip)* to free the memory pointed to by *ip* (see *free(3)*). If *ip* is the last inset pointing to its associated data object, *free(ip)* will also cause the data object to be destroyed; otherwise the data object will continue to exist.

Usage. You should refer to *BE2 Basics*, pp. 11 - 79 for examples of how to work with the *x_Destroy* construct.

Example. If you were defining an inset named *doc* with a *text* data object, you might want to create a procedure called *doc_Destroy (doc_pointer)*.

Destroying an inset

```
inset_Destroy (ip)
struct inset ip;
```

Description. *inset_Destroy* looks for an *x_Destroy* routine for the inset pointed to by *ip* and, if found, calls it; else it calls *free (ip)* in order to free the memory pointed to by *ip* (see *free(3)*). If *ip* is the last inset pointing to its associated data object, the data object will be destroyed; otherwise the data object will continue to exist. In addition, *inset_Destroy* forces a full update, so that all windows are cleared and redrawn.

Usage. You should always use the *inset_Destroy* procedure to destroy an inset of type *x*, rather than calling *x_Destroy* directly.

Communicating from the parent to the inset

An inset's parent is responsible for notifying an inset when it needs to redraw itself, adjust it's menus to a new window size, when it is the input focus and thus needs to take care of keyboard or mouse input, etc. The parent routine does this through routines described in this section.

Providing an full update for inset x

```
x_FullUpdate (ip, how, r)
struct inset ip;
int how;
struct rect r;
```

Description. If you are defining an inset named *x*, you must provide the routine *x_FullUpdate*. You must declare *x_FullUpdate* with three parameters: *ip*, the pointer to inset; *how*, how the window should be redrawn, and *r*, the rectangle within the inset that should be redrawn.

The how parameter specifies the degree of changes and can have the following values:

inset_FULLREDRAW -- the inset should be completely redrawn.
inset_PARTIALREDRAW - the inset should be redrawn within the rectangle specified by the parameter *r;* further partial redraws will follow.
inset_LASTPARTIALREDRAW - the inset should be redrawn within the rectangle specified by the parameter *r*; this is the last partial redraw.
inset_REMOVE - the inset in being removed from the screen.

In addition to redrawing the screen, your *x_FullUpdate* procedure must reset any regions that your inset is handling. Your routine may assume that

1. Your inset's parent has set the inset's rectangle and window appropriately.
2. Your *x_Update* routine will never be called before a call has been made to your *x_FullUpdate* routine.
3. Your inset's window has been selected.
4. Your inset's clipping region has been set to the part of its rectangle that is visible on the screen.
5. If your inset is appearing in a new window, the parent will call your inset's menu routines to add menus.

You must program your inset so that these assumptions will hold for any of its children as well (see *inset_FullUpdate*, p. 97).

Usage. You should refer to *BE2 Basics*, pp. 11 - 87 for examples of how to work with the *x_FullUpdate* construct. In general, if the inset is going to be static, the *x_FullUpdate* routine could ignore the *how*

parameter and always do a full redraw of the inset. The *how* parameter is provided to give the programmer the ability to optimize the *x_FullUpdate*.

Notifying an inset that it should do a full update

```
inset_FullUpdate(ip,how,r)
struct inset in;
int how;
struct rect r;
```

Description. *inset_FullUpdate* sets up the inset *ip* so that assumptions it makes when doing a full update are true, then calls the inset's *FullUpdate* routine. The parameter *how* specifies the degree of change and can have the following values:

inset_FULLREDRAW -- the inset should be completely redrawn.
inset_PARTIALREDRAW - the inset should be redrawn within the rectangle specified by the parameter *r;* further partial redraws will follow.
inset_LASTPARTIALREDRAW - the inset should be redrawn within the rectangle specified by the parameter *r*; this is the last partial redraw.
inset_REMOVE - the inset in being removed from the screen.

If the parameter *how* has the values inset_PARTIALREDRAW or inset_LASTPARTIALREDRAW, the parameter *r* is the rectangle within the inset that should be redrawn; else *r* is NULL.

1. If the inset *ip* has a parent, *inset_FullUpdate* sets the inset's window to its parent's window.
2. It selects the inset's window, making it the current window.
3. It sets the clipping rectangle to the part of the inset's rectangle that is visible on the screen.
4. It selects the inset's menu region id, making the inset the currently selected menu region.
5. It sets the character shim and space shims to NULL.
7. It defines the inset's menu region to correspond to the visible portion of the inset.
6. If the inset is appearing in a new window, it calls the inset's menu routines, if any. Then it calls *im_AddGlobalMenus* to update the global menus, if any.
8. It calls the inset's *x_FullUpdate* routine with parameters *ip*, *how* and *r*.

Side Effects. Sets the *im* clip rectangle to the inset's clip rectangle. Sets the *im* character shims and space shims to 0. Sets the current menu region to the inset's region id.

Usage. You should always use the *inset_FullUpdate* procedure to

cause an inset *x* to do a full update, rather than calling *x_FullUpdate* directly. Before calling *inset_FullUpdate* for a child inset, you must insure that the child inset's visible rectangle, *vr*, (see *struct inset* in *inset.h*, p. 91). is correctly defined. There are two cases: (1) the child inset's rectangle, *r*, is the same as your rectangle, i.e., the parent's rectangle, *r*; and (2) the child has a different rectangle.

If the child inset's rectangle , *r*, is the same as yours, then simply set the child inset's visible rectangle, *vr*, to your visible rectangle, *vr*. For example,

```
struct inset *parentinset, *childinset;
childinset->vr = parentinset->vr;
```

This is essentially what *im_Interact* does for the top level inset, where *vr* is equal to the dimensions of the window.

If you have assigned a different rectangle to the child inset, you must insure that the child's visible rectangle will be on the screen. The easiest way to insure this is to call *inset_ClipVisualRectangle (childinset, parentinset)*, which will set the child's visible rectangle to the result of clipping the child's rectangle, *r*, against your visible rectangle (See p. 99).

Because of its side effects, it is best for a parent inset to do all its own update drawing before calling *inset_FullUpdate* for its child insets. If it does not, then you will need to reset the *im* state parameters that *inset_FullUpdate* clobbers.

Clipping a child's visible rectangle

```
inset_ClipVisualRectangle (childinset, parentinset)
struct inset *childinset, *parentinset;
```

Description. *inset_ClipVisualRectangle* sets the *childinset*'s visible rectangle to the result of clipping the child's rectangle, *r*, against the *parentinset*'s visible rectangle, *vr*.

Usage. If you are working with child insets, you must insure that the child's visible rectangle is set correctly before any *inset_FullUpdate*. The *inset_ClipVisualRectangle* routine is designed to make this easy. *inset_ClipVisualRectangle* handles the following cases:

1. The inset's rectangle and visible rectangle are completely contained with the inset's window (or parent inset's visible rectangle).

2. The inset's window (or parent inset's visible rectangle) and visible rectangle are completely contained within the area of the inset's rectangle.

3. The inset's window (or parent inset's visible rectangle), visible rectangle and rectangle intersect.

(For illustration, zip
/cmu/unix/itcsrc/itc/insets/documentation/be2docs/Inset.vrclipcases.zip)

Providing an update routine for inset x

x_Update (ip)
struct inset ip;

Description. If you are defining an inset named *x*, you may want to provide the routine *x_Update*. You must declare *x_Update* with a single parameter, *ip*, a pointer to the inset. Whereas *x_FullUpdate* gets called when a inset's *window* changes and the screen needs to be updated, *x_Update* should be called when the inset's *data object* changes and the screen needs to be updated.

Your *x_Update* routine may assume that

1. The screen is correct as of the previous call to this routine or *x_FullUpdate*.
2. Your *x_Update* routine will never be called before a call has been made to your *x_FullUpdate* routine, so the inset's rectangle and window are properly set.
3. Your inset's window has been selected.
4. Your inset's clipping region has been set to the part of its rectangle that is visible on the screen.

You must program your inset so that these assumptions will hold for any of its children as well (see *inset_Update*, p. 101).

Usage. You should refer to *BE2 Basics*, pp. 11 - 79 for examples of how to work with the *x_Update* construct. If you do not provide a x_Update, then you must arrange to do an *x_FullUpdate* whenever the inset's data object changes and the screen needs to be updated.

Notifying an inset that it should do an update

```
inset_Update(ip)
struct inset ip;
```

Description. *inset_Update* sets up the inset *ip* so that assumptions it makes when doing an update are true, then calls the inset's *Update* routine. In particular,

1. It selects the inset's window, making it the current window.
2. It sets the clipping rectangle to the part of the inset's rectangle that is visible on the screen.
3. It selects the inset's menu region id, making the inset the currently selected menu region.
4. It sets the character shim and space shims to NULL.
5. It sets the inset's changed flag to NULL.
6. If there is an *x_Update* routine, it calls it with parameter *ip*.

Side Effects. Sets the *im* clip rectangle to the inset's clip rectangle. Sets the *im* character shims and space shims to 0. Sets the current menu region to the inset's region id.

Usage. You should always use the *inset_Update* procedure to cause an inset *x* to do an update, rather than calling *x_Update* directly.

Because of its side effects, it is best for a parent inset to do all its own update drawing before calling *inset_Update* for its child insets. If it does not, then you will need to reset the *im* state parameters that *inset_Update* clobbers.

Providing a way for inset x to add menus

```
x_AddMenus(ip)
struct inset ip;
```

Description. If you are defining an inset named *x*, and you want the inset to have its own menus, you must provide the routine *x_AddMenus*. You must declare *x_AddMenus* with a single parameter, *ip*, a pointer to the inset. You should place all your window manager menu calls in *x_AddMenus*. In the underlying window manager, menus should be associated with each window. Whenever the window changes, the menus associated with each inset in the window must be reset. *inset_FullUpdate* will call *x_AddMenus* directly whenever the window changes.

Usage. You should refer to *BE2 Basics*, pp. 11 - 79 for examples of how to work with the *x_AddMenus* construct.

Providing a way for inset x to work with keyboard input

```
int x_KeyIn(ip, ch)
struct inset ip;
int ch;
```

Description. If you are defining an inset named *x*, and you want it to be receive menu or keyboard input, you must provide the routine *x_KeyIn*. You must declare *x_KeyIn* with two parameters: *ip*, a pointer to the inset, and *ch*, the character which has been sent to your inset. Your inset will receive characters when it is the current input focus. Your *x_KeyIn* routine should parse any sequence of characters it is sent and return the following:

inset_KEYACCEPTABLE -- if it accepts the key.
inset_KEYUNACCEPTABLE -- if the key is unacceptable.
inset_KEYPARTIALACCEPT -- if the the key is possibly acceptable, i.e., if it is part of a keystroke command sequence .

Finally, your *x_KeyIn* routine should test whether the value of *ch* is inset_KEYSTATERESET. If the value of *ch* is inset_KEYSTATERESET, then your *x_KeyIn* routine should reset it's parse state to the beginning of a command sequence. It is a way to set up a cancel mechanism to allow the user to cancel out of a keystroke command sequence. Of course, if your inset does not handle sequences of keystrokes (and thus never returns a value of inset_KEYPARTIALACCEPT), it can ignore values of inset_KEYSTATERESET for *ch*.

Usage. You should refer to **BE2 Basics**, pp. 11 - 79 for examples of how to work with the *x_KeyIn* construct.

Notifying an inset that it has received keyboard input

```
int inset_KeyIn(ip, ch)
struct inset ip;
int ch;
```

Description. If there is an *x_KeyIn* routine defined for the inset *ip*, *inset_KeyIn* calls it with the parameters *ip*, the inset, and *ch*, the character to send the inset *ip*.

Return value. If *x_KeyIn* is defined, it returns whatever value x_KeyIn returns; else it returns NULL.

Usage. You should always use the *inset_KeyIn* procedure to cause an inset *x* to handle keyboard input, rather than calling *x_KeyIn* directly.

Providing a way for inset x to receive the input focus

```
x_ReceiveInputFocus(ip)
struct inset ip;
```

Description. If you are defining an inset named *x* and want the user to be able to do keyboard input to the inset, you must provide the routine *x_ReceiveInputFocus*. When the user chooses your inset as the input focus, *im_Interact* will call *x_ReceiveInputFocus*. You must declare *x_ReceiveInputFocus* with a single parameter, *ip*, a pointer to the inset.

In order to give the user feedback, your *x_ReceiveInputFocus* routine should highlight (1) the inset and (2) the precise point of input within the inset in some fashion. Then it should call a routine to get keystrokes from the user.

It is possible that you will want your *x_ReceiveInputFocus* to call *inset_ReceiveInputFocus* for one of its children. If you do, you should first get the keystrokes, then you should call *inset_ReceiveInputFocus* for the child inset, and then pass the keystrokes down to the child inset via the *inset_KeyIn* procedure. In this way, your inset can trap keystrokes prior to its child seeing them.

If you do not provide an *x_ReceiveInputFocus* routine for an inset *x*, the following default will be used:

```
default_ReceiveInputFocus(ip)
struct inset *ip;
{
    if (! ip->hasinputfocus)  {
        ip->hasinputfocus = 1;
        inset_WantUpdate(ip,ip);
    }
}
```

In this case, the *hasinputfocus* flag is set and the inset automatically requests to be Updated.

Usage. You should refer to *BE2 Basics*, pp. 11 - 79 for examples of how to work with the *x_ReceiveInputFocus* construct. Note that Input Focus only applies to characters that are sent from the user's keyboard; mouse events and menu selections always follow the mouse cursor.

Notifying an inset that it has received the input focus

```
inset_ReceiveInputFocus(ip)
struct inset ip;
```

Description. If there is a *x_ReceiveInputFocus* defined for an inset, *inset_ReceiveInputFocus* calls it.

Usage. If the user clicks within an inset's rectangle, the parent should notify the inset that it has the input focus by calling *inset_Receive InputFocus*. You should always use the *inset_ReceiveInputFocus* procedure to cause an inset *x* to notify an inset that it has the input focus, rather than calling *x_ReceiveInputFocus* directly.

Providing a way for inset x to give up the input focus

```
x_LoseInputFocus(ip)
struct inset ip;
```

Description. If you are defining an inset named *x*, and x will be receiving the keyboard input from the user, you must provide the routine *x_LoseInputFocus*. You must declare *x_LoseInputFocus* with a single parameter, *ip*, a pointer to an inset.

This routine will be called when x no longer has the input focus. x_LoseInputFocus should de-highlight the inset. If the inset has passed the input focus down to one of its children, then it should call inset_LoseInputFocus with the child's inset as an argument in order to notify the child that it has lost the input focus.

If an inset does not provide a LoseInputFocus routine the following default will be used:

```
default_LoseInputFocus(ip)
struct inset *ip;
{
    if (ip->hasinputfocus)  {
        ip->hasinputfocus = 0;
        inset_KeyIn(ip, inset_KEYSTATERESET);
        inset_WantUpdate(ip,ip);
    }
}
```

In this case, the *hasinputflag* is cleared, the insets key state is reset to the null state and the inset requests to be Updated.

Usage. You should refer to *BE2 Basics*, pp. 11 - 79 for examples of how to work with the *x_LoseInputFocus* construct.

Notifying an inset that it has lost the input focus

```
inset_LoseInputFocus(ip)
struct inset ip;
```

Description. If there is a *x_LoseInputFocus* defined for an inset, *inset_LoseInputFocus* calls it.

.

Usage. If the user clicks outside an inset's rectangle, the parent should notify the inset that it has lost the input focus by calling *inset_LoseInputFocus*. You should always use the *inset_LoseInputFocus* procedure to notify an inset that it has lost the input focus, rather than calling *x_LoseInputFocus* directly.

Providing a desired size routine for inset x

```
x_DesiredSize(ip, width, height, pass, desiredwidth,
      desiredheight, attributes)
struct inset ip;
int width, height, pass, *desiredwidth, *desiredheight,
      *attributes;
```

Description. If you are defining an inset named *x*, you may want to provide the routine *x_DesiredSize*. *x_DesiredSize* takes six parameters.

The parent inset provides the first three parameters to the child. The *width* and *height* specify a proposed width and height for the inset pointed to by *ip*.

The parameter *pass* can take on one of three values:

NULL - if both dimensions are flexible.
WIDTHSET - if the width can not be changed.
HEIGHTSET - if the height can not be changed.

In response to these three parameters, *x_DesiredSize* should set three values. The *desiredwidth* and *desiredheight* should be set to the size that *ip* wants to be.

The *attributes* parameter is set to a OR-ed combination of any of the following four values:

WIDTHSMALLER -the width could be made smaller
WIDTHLARGER - the width could be made larger
HEIGHTSMALLER - the height could be made smaller
HEIGHTLARGER - the height could be made larger.

or NULL, which indicates that the inset really wants to be the size it is

requesting.

The process of negotiating between the parent's proposal and the child's request will normally take place during the call to the *x_FullUpdate* of the parent. The actual size and position of the inset *ip* will be set prior to the parent calling the *x_FullUpdate* procedure of *ip.*

If an inset does not provide a DesiredSize routine the following default will be used:

```
default_DesiredSize(ip, width, height, pass,
    desiredwidth, desiredheight, attributes)
struct inset *ip;
long width;
long height;
long pass;
long *desiredwidth;
long *desiredheight;
long *attributes;
{
    *desiredwidth = width;
    *desiredheight = height;
    *attributes = NULL;
}
```

Usage. You should refer to *BE2 Basics*, pp. 11 - 79 for examples of how to work with the *x_DesiredSize* construct.

Negotiating the size of an inset

```
inset_DesiredSize(ip, width, height, pass,
      desiredwidth, desiredheight, attributes)
struct inset ip;
int width, height, pass, *desiredwidth, *desiredheight,
      *attributes;
```

Description. If there is a *x_DesiredSize* defined for an inset, *inset_DesiredSize* calls it.

When you make the call, *width* and *height* should be set to specify a proposed width and height for the inset pointed to by *ip*.

The parameter *pass* should be set to one of three values:

NULL - if both dimensions are flexible.
WIDTHSET - if the width can not be changed.
HEIGHTSET - if the height can not be changed.

After the procedure call, *desiredwidth* and *desiredheight* will be set to the size that *ip* wants to be. If *x_DesiredSize* did not exist, they will be set to *width* and *height*.

The *attributes* parameter will be set to a OR-ed combination of any of the following four values:

WIDTHSMALLER -the width could be made smaller
WIDTHLARGER - the width could be made larger
HEIGHTSMALLER - the height could be made smaller
HEIGHTLARGER - the height could be made larger.

If *x_DesiredSize* did not exist, *attributes* will be set to the NULL.

Usage. You should always use the *inset_DesiredSize* procedure to notify an inset that it has lost the input focus, rather than calling *x_DesiredSize* directly.

Providing a routine to handle mouse hits for inset x

```
struct inset *x_Hit(ip, action, x, y)
struct inset ip;
int action, x, y;
```

Description. If you are defining an inset named *x*, you must provide the routine *x_Hit*. You must declare *x_Hit* with four parameters *ip*, a pointer to the inset; *action*, the type of mouse hit that occurred, decodable by the underlying window manager; *x*, the horizontal location of the mouse cursor at the time of the hit; and *y*, the vertical location. *x_Hit* will be called when a mouse hit occurs within the rectangle administered by the inset.

In general, *x_Hit* should decode the action of the mouse hit and take appropriate action, or if it believes that the hit actually belongs to a child inset, it should call *inset_Hit* on the child inset.

Return value. If the inset itself handles the mouse hit, *x_Hit* should return a pointer to the inset; if it has passed the mouse hit to one of its children, it should return the value returned by its child's *x_Hit* routine so that the inset manager will know what inset gets the next mouse event: The inset manager passes all *DownTransitions* down the inset tree. After a *DownTransition* and until the *UpTransition*, all mouse events (i.e., *DownMovements*) will go to the inset that actually handled the initial *DownTransition*. The *DownMovements*, of course, can always be filtered further down the inset tree (from parent to child).

If an inset does not provide a Hit routine the following default will be used:

```
struct inset *default_Hit(ip, x, y, action)
struct inset *ip;
{
    return ip;
}
```

Notifying an inset that a mouse hit has occurred in its rectangle

```
struct inset *inset_Hit(ip, action, x, y)
struct inset ip;
int action, x, y;
```

Description. If there is a *x_Hit* defined for inset *ip*, *inset_Hit* calls it.

Usage. You should always use the *inset_Hit* procedure to notify an inset that it has received a mouse hit, rather than calling *x_Hit* directly.

Communicating from an inset to its parent

The following routines are called by the inset to request an action of its parent.

Providing a way to an inset x to request an update

```
x_WantUpdate(ip, descendant)
```

Description. If you are defining an inset named *x*, you must provide the routine *x_WantUpdate*. You must declare *x_WantUpdate* with two parameters: an inset *ip* and a child of the inset, *descendant*. *x_WantUpdate* will be called when the *descendant* inset, which must be a child of the inset *ip*, wants to be updated.

If an inset does not provide a *x_WantUpdate* routine the following default will be used:

```
default_WantUpdate(ip, descendant)
struct inset *ip;
struct inset *descendant;
{
    if (ip == descendant && ! ip->changed)
        inset_WantUpdate(ip->parent, descendant);
}
```

Usage. You should refer to *BE2 Basics*, pp. 11 - 79 for examples of how to work with the *x_WantUpdate* construct.

Requesting an update

```
inset_WantUpdate(ip, descendant)
struct inset ip, descendant;
```

Description. If there is an *x_WantUpdate* routine, it calls it with *ip*, the parent inset, and *descendant*, the child inset, as arguments; else if the inset *ip* has a parent, it calls *inset_WantUpdate* with the parent and descendant.

Usage. If an inset wants to request that it should be updated, it should call inset_WantUpdate with both arguments set to itself. You should always use the *inset_WantUpdate* procedure to notify a parent inset that an inset needs an update, rather than calling *x_WantUpdate* directly.

Providing a routine for requesting the input focus

```
x_WantInputFocus(ip, descendant)
struct inset ip, descendant;
```

Description. If you are defining an inset named *x*, you may want to provide the routine *x_WantInputFocus*. It will be called when the *descendant* inset desires the input focus.

If an inset does not provide a *x_WantInputFocus* routine the following default will be used:

```
default_WantInputFocus(ip, descendant)
struct inset *ip;
struct inset *descendant;
{
    if (ip->parent) inset_WantInputFocus(ip-
>parent, descendant);}
```

Usage. You should refer to *BE2 Basics*, pp. 11 - 79 for examples of how to work with the *x_WantInputFocus* construct.

Requesting the input focus

```
inset_WantInputFocus(ip, descendant)
struct inset ip, descendant;
```

Description. If there is an *x_WantInputFocus* routine, it calls it with *ip*, the parent inset, and *descendant*, the child inset, as arguments; else if the inset *ip* has a parent, it calls *inset_WantInputFocus* with the parent and descendant.

Usage. You should always use the *inset_WantInputFocus* procedure to notify a parent inset that an inset needs the input focus, rather than calling *x_WantInputFocus* directly.

Along with the inset_routines, the standard inset package provides the following:

```
inset_InitStructure(name, ip)
```

This takes an inset structure and its name and generates a list of standard procedures. It will put in defaults for procedures that are not provided and sets the value of a routine to NULL if there is no proper default.

Inset/Window Management Routines

The following routines are the inset-window management routines available in BE2. The routines in this package handle the interface between a top level inset and the underlying window management system. Thus, a programmer can create a window by calling the underlying window management system, and associate an inset with that window, as well as create and control menus. Note that the handle used in communication with this package is not a pointer to an inset, but rather a pointer to a window.

All inset/window management routines are prefixed by *im* .

Creating and deleting window insets

Creating a window to be used by an inset

```
struct im_window
*im_CreateWindow (host)
char *host;
```

Description. *im_CreateWindow* creates a window that can be used by the *im_FillWindow* routine which associates an inset with the window. The parameter host is the name of the host machine that is passed to the underlying window management system so it can determine what machine the window will be brought up on.

Return value. If successful, *im_CreateWindow* should return a pointer to a new window.

Usage. If you want the window to be brought up on the normal host, pass in NULL (or 0) for the host value. Passing nothing in for this argument will lead to random program behavior.

Deleting a window

```
im_DeleteWindow (win)
struct im_window   *win;
```

Description. If you have created a window, you can use this routine to delete it and its corresponding window. The window itself is deleted, but any inset visible in the window will not be destroyed. The parameter *win* is a pointer to the window to be deleted.

Usage. Since the routine does not call *inset_Destroy* automatically, if you want any inset visible in the window to to deleted to be destroyed you should call *inset_Destroy* specifically.

Putting an inset in a window

```
im_FillWindow(win, in)
struct im_window   *win;
struct inset in;
```

Description. This procedure associates a specific inset with a window and places the inset in the window created with *im_CreateWindow*. The parameters *win* and *in* specify the window to be used and the inset tc be placed in it, respectively. All mouse and keyboard events occurring in the window will be directed to the inset in the window.

Communicating from the parent to the window inset

Updating the windows

```
im_ForceUpdate ()
```

Description. This routine forces an update on all the windows and insets that have requested they be redrawn via *inset_WantUpdate*, by calling the procedure *inset_Update*.

Usage. See the section on **Inset Routines** pp.109-111 for complete descriptions of the *inset_WantUpdate* and *inset_Update* procedures.

Doing a full update on all the windows

```
im_ForceFullUpdate (win)
struct im_window  *win;
```

Description. This procedure causes window to be cleared and redrawn by calling on the procedure *inset_FullUpdate*. The parameter *win* indicates the particular window to be redrawn; if *win* is NULL, all windows will be fully redrawn.

Usage. See the section General Inset Routines pp. 96 - 98 for a full description of the *inset_FullUpdate* procedure.

Interacting with the outside

An interaction loop routine

```
im_Interact(mayBlock)
int mayBlock;
```

Description. im_Interact is an interaction routine that performs one of the following "user-interactions" and then returns.

Performs one character event of a keyboard macro.

Handles a redraw request from the underlying window management system,

Reads a character from a window and calls the appropriate *inset_KeyIn* or *inset_Hit* procedures, depending upon whether the character represents keyboard input or a mouse hit.

Handles input available from an arbitrary file descriptor.

Executes an event from the timer event queue.

Return Value. The procedure returns a boolean: FALSE if the *Interact loop* should be performed; TRUE if loop should end.

Usage. Since the procedure returns after performing a single operation, it must be called from a while loop in order to accomplish what in Unix/Emacs terms is called a recursive edit.

Handling an arbitrary file descriptor

```
im_AddFileHandler(f, handler, param, priority)
FILE   *f;
int   (*handler)  ();
char   *param;
int   priority;
```

Description. One of the operations that *im_Interact* can perform is calling an input handler for a file descriptor from which input is sought. Such an operation is enabled by calling *im_AddFileHandler*, which associates a function to be called with the file descriptor when input is available.

The parameter *f* is the pointer to a file in which the caller is interested. *handler* is the address of a procedure to call when input appears from the file descriptor. *param* will be passed to the procedure provided by *handler*, when it is called. The procedure provided by *handler* is called with two parameters, the first being the file descriptor upon which input is available, and the second being a parameter associated with this call to *im_AddFileHandler*. The last parameter is a priority. BE2 can only remember a fixed number of handlers, and the ones that it keeps are those associated with the smallest-numbered *priority*.

Return Value. Returns a boolean: TRUE if the procedure was successful in associating a function to be called with the file descriptor; FALSE otherwise.

Removing a handler

```
im_RemoveFileHandler ( f )
FILE *f;
```

Description. This procedure removes one handler for the named file descriptor from the list of handled file descriptors. The parameter *f* is a pointer to the file descriptor.

Data Object Routines

The following routines are the standard routines that must be provided by a data object. There are two groups of routines: (1) *x_* routines that you must provide when you define a data object for the inset named *x* and (2) *data_* routines, located in *data.c*, that you must use to call upon a particular inset's *x_* routines. The discussion divides the routines into the following groups:

Creating and deleting data objects
Reading and writing routines
Viewing routines

For each *x_* routine, there is a corresponding *data_* routine or macro. The *data_* routine typically sets up the inset so that its information is current, and then calls the corresponding *x_* routine. You should always use the *data_* routines to call the *x_* routines for two reasons: First, it is easier for you to call an an *data_* routine than to call the *x_* routine, because you don't need to manage the set up. Second, it is easier to define an inset *d* if it can assume that certain conditions will be true when it is called.

Creating and deleting data objects

Providing a creation routine for a data object

```
struct data *data_NewData (name)
char *name;
```

Description. If you are defining a data object named *d*, you must provide the routine *data_NewData*. You must declare *data_NewData* with a single parameter, *name*, the name of the new data object.

Return value. If successful, *data_NewData* should return a pointer to the struct *data* that it has created; NULL for failure.

Initializing a data object structure

```
data_InitStructure (name, dop)
char *name;
register struct data *p;
```

Description. This routine takes a data object structure and the name and generates the list of standard procedures. The routine puts in the defaults for procedures that are not provided and sets the value of the routine to a null routine if there is no proper default.

Deleting a data object

```
data_FreeData (dop)
register struct data *dop;
```

Description. This routine frees the storage held by the data object given by *dop*. It should recursively free any data objects contained within it.

Reading and writing routines

These routines are called by a data object to act on another data object.

Reading a data object

```
struct data *data_Read (dop, d,start,length)
register struct data *dop;
char *d;
int start, length;
```

Description. This routine reads the ascii representation of the appropriate data object, modifying the data object given by *dop*. If the routine encounters the representation of another data object,it will recursively call another data object's *read* routine. When it calls on the child to read its description, the parent data object does not need to understand the data the child is reading.

Writing to a data object

```
data_Write (dop, d, start, length)
register struct data *dop;
char *d;
int start, length;
```

Description. This routine takes a pointer to a data object *dop*, and writes the ascii representation of the data object to the file. It should recursively call *data_Write* on any objects in encounters recursively.

Finding out if a data object has been modified

```
data_GetModified (dop)
register struct data *dop;
```

Description. Since data objects can be read and written by other data objects, you may need to keep tabs on modifications made to a data object. This procedure indicates whether the data object has been modified since the last time it was written.

Return value. Boolean. True if data object was modified, False otherwise.

Usage. If a data object does not provide a *GetModified* routine, the following default will be used:

```
default_GetModified(dop)
struct data *dop;
{
        return (dop->modified != 0);
}
```

Setting a flag on a data object if the flag was modified

```
data_SetModified (dop, flag)
register struct dat *dop;
int flag;
```

Description. You may want the flag for a data object set to a certain value, despite changes made to it. This routine sets the flag for the data object specified by the pointer *dop* to the value *flag*.

Usage. If a data object does not provide a *SetModified* routine, the following default will be used:

```
default_SetModified(dop, flag)
struct data *dop;
int flag;
{
        dop->modified = flag;
}
```

Viewing routines

Viewing routines are used to associate an inset with a data object.

Adding an inset to a list of insets viewing the data object

```
data_AddViewer (data, inset)
struct data *data;
struct inset *inset;
```

Description. This routine adds the inset *inset* to the list of insets that are viewing the data object *data*. When an inset is added to this list it is requesting that the data object calls its *WantUpdate* routine whenever the data object is changed.

Usage. If the data object does not provide an *AddViewer* routine, the following default will be used:

```
default_AddViewer(dop, ip)
struct data *dop;
struct inset *ip;
int flag;
{
        struct insetlist *in;

        for (in = dop->viewers;
                in != NULL && in->ip != ip;
                in = in->next);

        if (in == NULL)     {
                in = (struct insetlist *)
                        malloc(sizeof (struct insetlist));
                in->ip = ip;
                in->next = dop->viewers;
                dop->viewers = in;
        }
}
```

Removing an inset from the list

```
data_RemoveViewer (data, inset)
struct data *data;
struct inset *inset;
```

Description. This routine removes the *inset* from the list of insets that are viewing the data object *data*.

Usage. The default routine is as follows:

```
default_RemoveViewer(dop, ip)
struct data *dop;
strucft inset *ip;
int flag;
{
        struct insetlist *in;
        struct insetlist *pin = NULL;

        for (in = dop->viewers;
                    in != NULL && in->ip != ip;
         in = in->next) pin = in;

        if (in != NULL)     {
         if (pin == NULL)
              dop->viewers = in->next;
              else
              pin->next = in->next;
              free(in);
        }
}
```

Finding the default inset for viewing the data object

```
data_InsetName(dop)
register struct data *dop;
```

Description. This routine returns the name of the default inset to be used for viewing objects of the type of the data object pointed to by *dop*.

Usage. The default routine is as follows:

```
char *default_InsetName(dop)
struct data *dop:
{
        return dop->dataobjectname;
}
```

Menu Routines

A BE2 application may control several windows. At any given time, each window has a set of available menu items. These items are available anywhere the mouse is clicked appropriately, i.e., middle-button click, within that window. The menu items available to a user is specified in an application by a MenuList. Menu lists are used by insets to describe which items should appear in a menu and the kind of processing that should take place if a menu item is selected. The menu package is used by insets to define and manipulate menu lists.

Menu items are provided by the insets in the window. Each inset may have any number of *menu lists*. Each list described the menu items an inset wants at a given time. Like everything else in BE2, the installation of menu items in a window is made for an inset by its parent. Whenever an inset wishes to provide a certain collection of menu items, it passes the list of the menu items to its parent. The parent is then responsible for making those menu items available to a user of a program (if deemed appropriate by the parent). Typically, a parent will just pass the list up to its parent, until the top of the inset tree is reached, where the inset manager will perform the actual window manager calls to install the menu items. An inset may decide to change its menus during execution. For example, a drawing editor may change the menus depending on what kind of object has been selected for manipulation. Whenever an inset desires the menus to change, it will call its parent with a different list of desired menu items.

The inset manager provides an interface for the "top-most" inset to use to install menus. The implementation of the interface will execute the necessary window manager calls for installing the desired menu items. BE2 applications should *never* make window manager calls directly for menu manipulation.

Menu definition and installation

The current menu facility separates menu definition and menu installation. Menu definition is the process of creating the available menu items. Menu installation is the process of making menu items visible (and selectable) by the user. Some applications may choose to combine these two processes, but we will try to keep them separate in our discussion.

An inset typically defines a menu when it is created and initialized. Typically, the *Init* procedure of an inset will create all the menu lists that it will ever use, and keep these menu lists as local variables. In many circumstances, it will be possible for all insets of the same type to share menu lists.

Empty menu lists are created by calling *menu_NewML* which returns a value of type *MenuList*. menu_NewML takes a single parameter, the inset that is building the menu list (usually, the inset making the NewML call). Items are added to the menu list by repeated calls of *menu_AddToML*, which takes four parameters:

MenuList: The menu list which to be appended with a new menu item;

menuString: A pointer to the string that is to be used to denote the menu item to the user;

proc: A pointer to the procedure that should be called when the menuItem is selected (see below)

rock: A long word worth of arbitrary data to be passed to proc when called (see below)

When a menu list is no longer needed, such as when the inset is about to be destroyed, then the menu list should be discarded by using the procedure *menu_FreeML*, which takes as its only parameter the menu to be reclaimed. Typically, this call will be made in the inset's *Destroy* procedure.

When a menu item is selected by the user, the procedure specified in the menu item will be called with the following two parameters:

inset : A pointer to the inset responsible for the menu item (as passed to NewML)

rock: A long word of arbitrary data. This is the datum that was specified in the menu_AddToML call for this menu item.

The selection of a menu does not, by itself, cause any changes in the available menus to the user, nor does it change the input focus.

Eventually a menu list must be presented to the inset manager (or its functional equivalent) where the list will be converted in appropriate window manager calls. Typically, the final menu list will be produced by the topmost inset. There are several paradigms that children insets can use to negotiate with their parents about menu lists. After discussing the protocol used for communicating menu lists between parent and children insets, we will consider two specific paradigms that can implemented with the protocol.

In all paradigms, the communication between parent and children insets about menu lists takes place through the *WantMenuList* procedure. This procedure must be provided by every inset (we will discuss the default procedure below). A typical call of this procedure to establish the menus desired by an inset would look like:

```
WantMenuList(parent,self,DesiredMenuList)
```

This should be read that the WantMenuList in the inset's parent (parent.WantMenuList) should be called with the parameters of the requester of the menu list change (self) and the menu list that should be used by the application (DesiredMenuList). This would be coded as:

```
inset_WantMenuList(ParentInsetPointer,CurrentInsetPointer,DesiredMenuList);
```

where *ParentInsetPointer* and *CurrentInsetPointer* are presumed to be

declared and initialized appropriately.

Focus-controls and universal menus

Two common paradigms that can be used by children and parents for manipulating menus are Focus-controls-menu and Universal-menus (also known as Menu-Chaos).

Focus-controls-menus uses the paradigm that the current focus of interaction (usually keyboard interaction), should control which menus are available. Therefore, when an inset receives the input focus, it should call the *WantMenuList* procedure of its parent, passing the menu list that it would like to have displayed. When the inset loses the input focus, it should call its parent with an empty menu list indicating that it no longer wants its menus to be displayed. (Note: an empty menu list is not a null -- an empty menu list is returned by a call to menu_NewML and must be freed, when no longer needed, by corresponding calls to menu_FreeML.) The menu list provided by the "focused" inset completely specifies the available menu items for the user -- no other menu items will be displayed.

In the focus-controls-menus paradigm, the parent of a focused inset would just pass the same menu list up to its parent without changing the "requester of the menu list change". This would correspond to the procedure:

```
procedure WantMenuList(RequestingInset,RequestedMenuList);
{
        WantMenuList(parent,RequestingInset,RequestedMenuList);
}
```

(Note: the first parameter is *RequestingInset*, not *self* as before.) The topmost inset would call the inset manager interface to the window manager to install the desired menus. (Note: when an inset loses the input focus, an empty menu list would be get passed up the inset tree unti lit reached the inset manager, which would cause all menu items to be removed.)

Universal-menus uses the paradigm that all menus for all insets should always be available. Menus come into existence when the inset is created and disappear when the inset disappears. The rest of the discussion assumes that containing insets are created by their parent inset when the parent inset is created, and they are destroyed by the parent inset when the parent inset is destroyed.

When an inset runs its *Init* procedure (after being created), it not only creates its menu list, but immediately calls its parent *WantMenuList* to request that the menu list be used. The parent will not call its parent *WantMenuList*. Instead, the parent will save the child's menu list in a local variable of the parent dedicated to that child. When the parent has created all of its children, it will have stored in local variables the menu list that each child requested. The parent will create its own list, concatenate all of the lists, and pass the new list to its parent by making a *WantMenuList* call. Concatenation of lists is accomplished by the *menu_ChainML* procedure. This procedure takes two

menu lists as arguments. The second chain will be logically appended to the first chain. Naturally, if a parent has only one child and no menu lists of its own, then it would just pass the child's menu list up to its parent. Like the focus-controls-menus paradigm, the topmost inset would pass the final list to the inset manager for translation into appropriate window manager calls.

Because we assume that an inset is destroyed by its parent, the parent knows that the menus need to be redefined, and will make calls to its parent after destruction of the child (after reconstructing a new concatenated menu list). Note: this implies that the WantMenuList procedure must distinguish between the time it is called for initial setup and the time it is called to update an already established collection of insets. One easy way to accomplish this goal would be to check if the requesting inset had already provided a desired menu list. If not, then the passed list is being provided as an initial value to be stored for later use. If a menu-list value already exists for that inset, then an update is being made and the reconstructed menu list should be immediately passed up the inset tree.

Of course, these two paradigms can be used in combination. For example, we might believe that the topmost inset may have some menu items with universal availability, e.g., Quit, while all other insets would have menus only on becoming "focused". When the focused inset's menu list reached the top most inset, the top most inset would concatenate its own list before passing the resulting menu list to the inset manager.

Another combination is that an inset may have one set of menus when it is focused and another when it is not. When an inset is focused, it passes its "active" menu list to its parent; when it loses the focus, it passes its "passive" menu list to its parent. However, the parent implementation of WantMenuList changes radically from the Focus-controls-menus paradigm. That paradigm had the parent just passing up the menu list (eventually to the inset manager). If that were done here, the inset manager would first cause the menus of the deactivated inset to be installed, then almost immediately, the activated inset's menu list would replace the deactivated inset's menu list by another call up the inset tree to the inset manager. Instead, each parent must collect the menu lists of its children, like in the universal-menu paradigm. The critical difference is when to call a parent's WantMenuList. The appropriate time is when the parent has finished executing its ReceiveFocus procedure, which would typically look like:

```
ReceiveInputFocus(parent) {
    call "correct" child's ReceiveInputFocus(self);
    /* some time here child calls WantMenuList which stores menu
list*/
    construct new menu list of activated and passive children, and
self;
    parent.WantMenuList;
}
```

Menu procedures

This section describes the menu package and its related routines.

Creating a menu list

```
MenuList *menu_NewML(OwnerInsetPtr)
inset *OwnerInsetPtr;
```

Description. This procedure creates an empty menu list pointer that is "owned" by the OwnerInset. The OwnerInsetPtr is passed to procedures called when menu items in the list are invoked.

Freeing a menu list

```
menu_FreeML(MenuListPtr)
MenuList * MenuListPtr;
```

Description. This procedure frees the designated menu list. Other menu lists that were prepended to this menu list now have undefined chains and need to be unchained.

Adding a menu to a menu list

```
menu_AddToML (MenuListPtr, MenuProc, menuString, rock)
MenuList *MenuListPtr;
char *MenuProc;
char *menuString;
long rock;
```

Description. This procedure adds a menu item to a menu list. The MenuProc is the procedure that will be called when the menu item is selected. The MenuString is the user visible presentation of the menu item. The rock is an long datum that will be passed to the MenuProc when the menu item is selected. If the string is aleady in the menu list, the MenuProc and rock associated with the string will be changed. The effects of adding a menu item with a string that is the same as another menu item in a chained menu list is undefined. This call changes the version number of the menu list.

Deleting a menu item from the menu list

```
menu_DeleteFromML (MenuListPtr, menuString)
MenuList *MenuListPtr;
char *menuString;
```

Description. This procedure deletes the menu item with the given menuString from the menu list. Chained lists are not examined. If no menu item has the string, then no change is made to the menu list. This call changes the version number of the menu list.

Clearing a menu list

```
menu_ClearML (MenuListPtr)
MenuList *MenuListPtr;
```

Description. Clears a menu list out completely. This call has no effect on chained menu lists. This call changes the version number associated with the menu list.

Connecting two menu lists

```
menu_ChainML(FirstMenuListPtr,NextMenuListPtr)
MenuList *FirstMenuList, *NextMenuListPtr;
```

Description. This procedure is used to logically connect two menu lists. The chain headed by NextMenuList is appended to the chain headed by FirstMenuList. Menu lists in either chain may be altered without affecting the chaining, i.e., one can add, delete or clear a menu list and the lists will still be chained. Freeing a menu list will cause the chaining of the menu lists to become illegal -- one should unchain lists first by calling UnlinkML(NextMenuListPtr) (see below). This does not changed the menu list's version number.

Positioning the pointer

```
menu_RewindML(ML)
MenuList *ML;
```

Description. Position the hidden menu list pointer to the start of the menu list.

Getting the next item from a menu list

```
long menu_NextME (ML, aproc, astring, arock)
MenuList *ML;
char **aproc;
char **astring;
long *arock;
```

Description. Get the next menu item from a menu list. Returns 0 for the menu string if no more menu items; returns some undefined non-zero value if the parameters were set. Note that calling menu_NextME repeatedly after a menu_RewindML is defined to show you the entire contents of the menu list.

Splitting a menu list chain

```
menu_SplitML(MenuListPtr)
MenuList *MenuListPtr;
```

Description. This procedure is used to logically split the chain containing the menu list denoted by MenuListPtr. The menu list before MenuListPtr will become the end of its chain, the rest of the chain will start with MenuListPtr. This does not change any menu list's version number.

Finding chained menu lists

```
MenuList  *menu_ChainedML(MenuListPtr)
MenuList *MenuListPtr;
```

Description. This procedure returns a pointer to the menu list that is chained (logcailly follows) the specified menu list. If no menu list is chained, then 0 is returned.

Changing the owner of a menu list

```
menu_OwnML(MenuListPtr,OwnerInset)
MenuList *MenuListPtr;
inset *OwnerInset;
```

Description. This procedure sets the owner of a menu list to the specified OwnerInset. This is as if the NewML had been called with OwnerInst. This does not change the version number.

Finding the owner or a menu list

```
inset *menu_MLOwner(MenuListPtr)
MenuList *MenuListPtr;
```

Description. This procedure returns the "owner" inset of the menu list.

Finding the version number of a menu list

```
long menu_GetMLVersion(ML)
MenuList *ML;
```

Description. This call returns the version number of a menu list. It can be used to check if the contents of a menu list are the same as at some other time.

Telling the inset you want a menu list

```
inset_WantMenuList (called_inset,requesting_inset, ml)
inset *called_inset, *requesting_inset;
MenuList *ml;
```

Description. Calls the WantMenuList procedure of the called inset with the other two parameters. Normally, the called_inset would be the parent inset and the requesting inset would be the inset calling inset_WantMenuList. The default procedure is that the called_inset is the parent inset and the other two parameters are passed up unchanged. See the discussion of inset communication for more information.

Getting user response

```
MenuItemHit(OwnerInsetPtr, Rock)
inset *OwnerInsetPtr;
long Rock;
```

Description. When a menu item is hit, a procedure with the above signature (parameters) is called. The OwnerInsetPtr is the owner that was specified in the menu list, and the rock is the long datum specified in the AddMenuItem call.

Interfacing between BE2 and the window manager

```
im_InstallML(MenuListPtr)
MenuList *MenuListPtr;
```

Description. This procedure provides an interface between BE2 and the underlying window manager. The top most inset responsible for menus should call im_InstallML with the menu list that should be used for the application.

The Keymap Package

The keymap package provides a set of facilities that will often make writing the *x_KeyIn* procedure for an inset *x* substantially easier. The keymap package allows you to bind sequences of keyboard characters to procedures. Typically, each procedures will be command procedure, i.e., the procedure will interpret the sequence of characters that the user types as a command. If you use the keymap package facilities to bind a sequence of characters to a procedure and set up your *x_KeyIn* accordingly, then when the user types that sequence, the *im_Interact loop* (see p. 115) will invoke the corresponding procedure. For example, if you bind *ctrl-x ctrl-s* to a procedure called *Write_Buffer_to_File*, then when the user types *ctrl-x ctrl-s*, the *im_Interact loop* will invoke *Write_Buffer_to_File*. Likewise, if you bind the keystroke *a* to *Insert_Character*, then when the user types an *a*, the *Im_Interact loop* will invoke the procedure *Insert_Character*. To use the keymap facilities, you must make calls to the keymap library to make the bindings and you must write the procedures.

Introduction to keymap facilities

Formally, a keymap is a function that maps a sequence of keyboard characters to a procedure. To understand how keymaps work, it is useful to think of a keymap as an array with 128 entries, one for each ASCII character. Each entry in the keymap array may have one of three values: a procedure, another keymap, or a special value used to indicate that the character has no binding.

Bindings

Procedure bindings

Whenever the user types a character, the character is used as an index into the keymap array. If the array value for the character is a procedure, then the mapping is done and the procedure is executed. For example, suppose the keymap *Edit_BasicCommands* maps *ctrl-s* to the procedure *Edit_SearchForward*. Then when the user types a *ctrl-s*, the character is mapped to the procedure *SearchForward*, which is then executed.

Sub-keymap bindings

If the array value for the character that a user types is another keymap, then the specified keymap will be used to map the next character that the user types. For example, suppose that the keymap *Edit_BasicCommands* maps *ctrl-x* to another keymap, *Edit_eXtendedCommands*, and that *ctrl-s* in the keymap *eXtendedCommands* maps to the procedure *WriteBuffertoFile*. Then when the user types *ctrl-x*, the keymap *eXtendedCommands* will be used to interpret the next character that the user types. If the user types *ctrl-s*, then that will map to the procedure *WriteBufferToFile*, which will be executed.

A keymap that itself has sub-keymaps defines an implicit tree, where one keymap is the root keymap, and the other sub-keymaps are branches.

No bindings

Finally, if the character that the user types maps to a value that indicates there is no binding, then the keymap returns a status code indicating that there was no procedure bound to that sequence of keys.

Keystates

If keymaps only mapped single keys to procedures, then there would be no need to keep track of the state of the mapping. To map *sequences* of keys to procedures, however, requires keeping state information. For instance, the mapping process must know which keymap is being used to evaluate the next incoming character. This state information is kept in a *keystate* that you must associate with the root keymap that you create.

Additional keymap facilities

Arguments

The keymap package allows you to define numeric arguments as part of a character sequence. Numeric arguments are typically used in order to allow the user to repeat a command easily or to allow the user to specify an alternative value for a command's parameter. For example, the *Document* package (see p. 151) defines *ctrl-u* as an argument procedure. If the user types the character sequence *ctrl-u 10 ctrl-f*, the procedure *Forward* executes 10 times, resulting in the text cursor moving forward 10 characters.

Last Command

The keymap package also provides facilities that allow you to keep track of the last command that the user has entered. This information is useful if the behavior of procedures you are writing needs to be depend upon previous procedures that the user has called. For example, the *Document* package (see p. 151) makes the behavior of *ctrl-n* and *ctrl-p* depend upon whether the previous character was a *ctrl-n* or *ctrl-p*.

Parent-Child parallel processing of keys

Finally, the keymap package allows you to let a parent inset and a child inset interpret a character in parallel, specifying which inset should take precedence if both the parent and the child want to accept the key. For example, if you are programming a stand-alone application inset, the top level inset provided by the *Inset Manager* maps *ctrl-c* to *exit*. But the *Inset Manager* sends the key to to the top level inset and to the your inset in parallel, giving you precedence. Thus, if you want to define *ctrl-c* yourself, you can, and your definition will prevail. On the other hand, if you are working with a child inset, you may want offer keys to both your inset and to the child, but override the child inset's interpretation in some cases. For example, if you use the *Document* package (see p. 151) to create a child inset, you may wish to override its mapping of *ctrl-s* to *view_search* and provide your own

search function. In that case, you would offer keys to your inset and the child document inset in parallel, but give your inset precedence.

Keymap routines

Creating a keymap

```
struct keymap
keymap_create()
```

Description. *keymap_create* creates a new keymap by dynamically allocating memory for a keymap structure. In addition, it initializes the newly allocated keymap structure so that all the keymap entries have the status keymap_EMPTY, that is, no procedures and sub-keymaps are bound to characters.

Return value. *keymap_create* returns a pointer to the newly created keymap.

Usage. If you are creating an inset named *x* and you want to use keymaps, then in the *x_Init* procedure for the inset, you should call *x_kmap = keymap_create ()* for each keymap that your inset needs. After calling *keymap_create*, you will use the pointers to bind characters to procedures and other keymaps.

Example.

```
struct keymap *Edit_BasicCommands;
struct keymap *Edit_eXtendedCommands;

Edit_BasicCommands = keymap_create();
Edit_eXtendedCommands = keymap_create();
```

creates two keymaps, EditText_Keymap and EditText_eXtendedCommands.

Binding a character to a procedure

```
keymap_insertproc (kmap, c, procedure)
struct keymap *kmap;
int c;
char *procedure;
```

Description. *keymap_insertproc* sets the keymap, *kmap,* to return *procedure* whenever the user types the character *c.*

Usage. If you are creating an inset named *x* and you want to use keymaps, then in the *x_Init* procedure for the inset, you should call *x_keymap_pointer = keymap_create ()* for each keymap that your inset needs. Then you should call *keymap_insertproc* as needed for the procedure entries that you want defined in your keymaps.

Example.

```
struct keymap *Edit_BasicCommands;
struct keymap *Edit_eXtendedCommands;

...

EditText_BasicCommands = keymap_create();
EditText_eXtendedCommands = keymap_create();

...

keymap_insertproc(Edit_BasicCommands, '\016', Edit_NextLine);
keymap_insertproc(Edit_BasicCommands, '\023', Edit_Search);
keymap_insertproc(Edit_eXtendedCommands, '\023',
Edit_WriteBufferToFile);
```

associates the procedure *Edit_NextLine* with the character *'\016,'* a *ctrl-n,* and the procedure *Edit_Search* with the character *'\023,'* a *ctrl-s* in the keymap *Edit_BasicCommands*; it associates the procedure *Edit_WriteBufferToFile* with *ctrl-s* in the keymap *Edit_eXtendedCommands.*

Binding a sequence of keys to a procedure

```
keymap_insertmap(kmap, c, sub-keymap)
struct keymap *kmap;
int c;
char *sub-keymap;
```

Description. *keymap_insertmap* associates the specified character, *c*, with the *sub-keymap* for the keymap, *kmap*. When the user types the specified character, the next character the user types will be interpreted by *sub-keymap*.

Usage. If you are creating an inset named *x* and you want to use keymaps, then in the *x_Init* procedure for the inset, you should call *x_keymap_pointer = keymap_create ()* for each keymap that your inset needs. Then, if you want to associate a sub-keymap with an entry in a keymap, you should call *keymap_insertmap*.

Example.

```
struct keymap *Edit_BasicCommands;
struct keymap *Edit_eXtendedCommands;

...

EditText_BasicCommands = keymap_create();
EditText_eXtendedCommands = keymap_create();

...

keymap_insertmap(Edit_BasicCommands, '\030',
Edit_eXtendedCommands);
```

associates the keymap *Edit_eXtendedCommands* with the character *'\030,'* a *ctrl-x*, in the keymap *Edit_BasicCommands*.

Binding a character to null

```
keymap_insertnull(kmap, c)
struct keymap *kmap;
int c;
```

Description. Associates a NULL operation with keymap *kmap*'s character, *c*.

Usage. When you create a keymap, all the characters in the keymap are associated with NULL operations by *keymap_create*. Thus, the only reason to call *keymap_insertnull* is if you want to rebind a key to a NULL operation.

Creating state information for a keymap

```
struct keystate
*keymap_newstate (kmap)
struct keymap *kmap;
```

Description. *keymap_newstate* allocates a new *keystate* for the keymap, *kmap*, and initializes it.

Return value. *keymap_newstate* returns a pointer to the newly allocated *keystate*.

Usage. If you are creating an inset named *x* and you want to use keymaps, then in the *x_Init* procedure for the inset, you should call *x_keymap_pointer = keymap_create ()* for each keymap that your inset needs. Then, you should call *keymap_insertproc* or *keymap_insertmap* as needed to bind procedures and sub-keymaps to character entries in the keymap. Finally, you should call *keymap_newstate* to create and initialize a keystate for the root keymap, and store the keystate with your inset.

Example.

```
struct Edit_inset {
     struct inset *insetp;
     struct keystate *kstate;
}Edit_insetp;

struct keymap *Edit_BasicCommands;
struct keymap *Edit_eXtendedCommands;

...

EditText_BasicCommands = keymap_create();
EditText_eXtendedCommands = keymap_create();

...

keymap_insertproc(Edit_BasicCommands, '\016', Edit_NextLine);
keymap_insertproc(Edit_BasicCommands, '\023', Edit_Search);
keymap_insertmap(Edit_BasicCommands, '\030',
       Edit_eXtendedCommands);
keymap_insertproc(Edit_eXtendedCommands, '\023',
       Edit_WriteBufferToFile);

...

Edit_insetp->kstate = keymap_newstate (Edit_BasicCommands);
```

Mapping sequences of keys

```
int
keymap_char(kstate, insetp, c)
struct keystate *kstate;
struct inset *insetp;
long c;
```

Description. *keymap_char* performs one step in mapping a sequence of characters to a procedure. It simulates the typing of the character *c* to the keystate *kstate*. If the *kstate* maps the character to a procedure, *keymap_char* calls the procedure with two parameters, *insetp* and *c*. If the *kstate* maps to a sub-keymap, *keymap_char* sets the *kstate* so that the next character that the user types will be mapped using the sub-keymap.

Return value. *keymap_char* returns inset_KEYACCEPTABLE if procedure is called, inset_KEYUNACCEPTABLE if no binding for the character sequence was found, and inset_KEYPARTIALACCEPT if it is in the middle of processing a key sequence.

Usage. If you have defined a keymap, then you should call *keymap_char* in your *x_KeyIn* procedure. Your *x_KeyIn* procedure should check the return value of *keymap_char*. If it is inset_KEYACCEPTABLE, you should call *inset_WantUpdate* for the inset.

Example.

```
x_KeyIn (insetp, c)
struct inset *insetp;
long c; {

   register struct x_inset *x_insetp = (struct x_inset *) (insetp);
   register int result;

   result = keymap_char (x_insetp->kstate, insetp, c);
   if (c == inset_KEYSTATERESET)
        return result;
   if (result = keymap_KEYACCEPTABLE)
        inset_WantUpdate (insetp, insetp);
   return result;
}
```

This example shows how to set up an inset for working with keymap's within *x_KeyIn*. Note that the example assumes that you have stored a pointer to the keystate (see *keymap_newstate*) in *x_insetp->kstate*.

Initializing a keystate

```
keymap_initstate(kstate, kmap)
struct keystate *kstate;
struct keymap *kmap;
```

Description. This procedure initializes the keystate *kstate* for the keymap *kmap*. If *kmap* is NULL, then *kstate* is reset to its initial state using the keymap with which it was previously associated.

Usage. *keymap_newstate* calls *keymap_initstate* to initialize a new keystate. Thus, the only reason to call *keymap_initstate* is to *re*-initialize a keymap's keystate.

Parallel processing of keys

If you are working with child insets, the keymap package allows you to let your inset and a child inset interpret a character in parallel, specifying which should take precedence if both you and the child want to accept the key. For example, the top level inset provided by the *Inset Manager* maps *ctrl-c* to *exit*. But the *Inset Manager* sends the key to to the top level inset and to the your inset in parallel, giving you precedence. Thus, if you want to define *ctrl-c* yourself, you can, and your definition will prevail. On the other hand, if you are working with a child inset, you may want offer keys to both your inset and child, but override the child inset's interpretation in some cases. For example, if you use the *Document* package (see p. 151) to create a child inset, you may wish to override its mapping of *ctrl-s* to *view_search*, providing your own search function. In that case, you would offer keys to your inset and the child document inset in parallel, but give your inset precedence.

Giving precedence to the inset

```
keymap_InsetAndKeymap(parent, child, kstate, c)
struct inset *parent, *child;
struct keystate *kstate;
long c;
```

Description. *keymap_InsetAndKeymap* offers the character *c* to the inset *child* by calling *keymap_char (kstate, child, c)* and offers the character *c* to the inset *parent* by calling *keymap_char (kstate, parent, c)*. If both the child and the parent have a binding for the character, the child's binding takes precedence.

Return value. *keymap_InsetAndKeymap* returns inset_KEYACCEPTABLE if a procedure is called, inset_KEYUNACCEPTABLE if no binding for the character sequence was found, and inset_KEYPARTIALACCEPT if it is in the middle of processing a key sequence.

Usage. If you have defined a keymap for your inset, but want a child inset to be able to override it with its own keymap, then you should call *keymap_InsetAndKeymap* rather than *keymap_char* when processing keyboard input.

Example.

```
return_code = keymap_InsetAndKeymap (ip, ip->inputfocus, ip-
      >kstate,c);
```

Giving precedence to the parent

```
keymap_KeymapAndInset(parent, child, kstate, c)
struct inset *parent, *child;
struct keystate *kstate;
long c;
```

Description. *keymap_KeymapAndInset* offers the character c to the inset *parent* by calling *keymap_char (kstate, parent, c)* and offers the character c to the inset child by calling *keymap_char (kstate, child, c).* If both the parent and the child have a binding for the character, the parent's binding takes precedence.

Return value. *keymap_KeymapAndInset* returns returns inset_KEYACCEPTABLE if a procedure is called, inset_KEYUNACCEPTABLE if no binding for the character sequence was found, and inset_KEYPARTIALACCEPT if it is in the middle of processing a key sequence.

Usage. If you have defined child insets, but do not want them to be able to override your keymaps, you should call *keymap_KeymapAndInset* when offering them keyboard input.

Argument facilities

The keymap package allows numeric arguments as part of a character sequence. Numeric arguments are typically used to allow the user to repeat a command easily or to allow the user to specify an alternative value for a command's parameter. For example, the *Document* package (see p.151) defines *ctrl-u* as an argument procedure. If the user types the character sequence *ctrl-u 10 ctrl-f*, the procedure *view_Forward* executes 10 times, resulting in the text cursor moving forward 10 characters.

If you are defining numeric arguments, you will want to associate a character sequence, like ctrl-u, that will act as a numeric argument command procedure. That procedure should be prepared to set the argument. Then the rest of your command procedures in the keymap will need to check whether the user has provided an argument.

The keystate uses an argstate structure and a set of procedures to keep track of arguments for you.

```
struct argstate {
     int argument;
     int argprovided;
     int argnext;
     int argdigit;
}
```

argument --The current argument. This is initialized to 1.
argprovided -- A boolean that indicates whether the user has provided an argument; initially FALSE.
argnext -- A boolean that indicates whether *argument_provided* should be set upon the next command.
argdigit -- A boolean that indicates that a digit has been already seen in this argument; initially FALSE.

Getting the argument state

```
struct argstate
*keymap_argstate ()
```

Description. keymap_argstate returns a pointer to the argument state for the keystate.

Usage. You should call keymap_argstate to get a pointer to the argument state. This is typically necessary if you need to access parts of the argstate structure for which there are no procedures defined.

Setting the argument

```
keymap_providearg(kstate, value)
struct keystate *kstate;
int value;
```

Description. *keymap_providearg* sets the keystate, *kstate*, so that the keymap package provides the argument *value* to the next command procedure that calls *keymap_argprovided*..

Usage. If you are defining a command procedure for numeric arguments, then the procedure should call *keymap_providearg* as appropriate. For example, the Document package defines *vcmd_ctrlu* as a numeric argument command procedure. If the user types ctrl-u ctrl-f, it means go forward 4 characters; if ctrl-u ctrl-u ctrl-f, go forward 64 characters.

Example.

```
vcmds_ctrlu (v)
    struct view *v; {
    register struct argstate *as;

    as=keymap_argstate();
    as->argdigit = 0;

    if (keymap_argprovided(v->kstate)
       keymap_providearg (as->argument * 4);
    else keymap_providearg (4);
}
```

Getting the argument

```
keymap_argument(kstate)
struct keystate *kstate;
```

Description. *keymap_argument* returns the argument provided by the keystate, *kstate*, or 1 if none was provided.

Usage. Each command procedure that you write should get the argument that the user specified and do the command the specified number of times. For example, the Document packages' routines all call *keymap_argument* to see how many times to execute.

Example.

```
vcmds_backword (v)
   register struct view *v; {

   register int i, argument_count;
```

```
    i = 0;
    argument_count = keymap_argument(v->ks);
    while (i < argument_count) {
          /* move back a single "word" in the document */
          i++;
    }
}
```

Clearing the command argument

```
keymap_cleararg(kstate)
struct keystate *ks;
```

Description. *keymap_cleararg* sets the *argument* associated with the keystate, *kstate*, to 1, and sets the keystate so that *keymap_argprovided* returns FALSE.

Usage. This procedure is intended to be invoked by a command procedure that needs to clear the argument state. Typically, if the command procedure is going to invoke another command procedure, the argument state should be cleared. For example, in the *Document* package, *vcmds_deleteword* invokes *vcmds_forwardword*, so it must clear the argument state before invoking it.

Example.

```
vcmds_deleteword (v)
   register struct view *v; {

   register struct int i, argument_count, pos;

   i = 0;
   argument_count = keymap_argument();
   keymap_cleararg();
   while (i < argument_count) {
       pos = view_getdotpos(v);
       vcmds_forwardword(v);
       doc_delete(view_document(v), view_getdotpos(v)-pos);
       view_setdotpos(v,pos);
       i++;
   } }
```

Testing whether there is an argument

```
keymap_argprovided(kstate)
struct keystate *kstate;
```

Description. *keymap_argprovided* returns TRUE if the user has already provided an argument for the keystate, *kstate*; FALSE otherwise. *keymap_argprovided* is a macro, so arguments should not have side effects.

Usage. If a command procedure that you are writing needs to test whether the user has provided an argument, then it should use this procedure. For example, the Document package's *vcmds_digit* procedure tests whether the user has typed the digit in the context of an specifying a command argument or in the context of wanting to insert a digit into the document.

Example.

```
vcmds_digit (v, c)
   struct view *v;
   char c; {

   register struct argstate *as;

   as = keymap_argstate ();
   if (keymap_argprovided(v->ks)) {
         if (as->argdigit == 0) {
             as->argdigit == 1;
        as->argument = 0;
         }
         keymap_providearg (as->argument*10+c-0x30);
   }
   else vcmds_SelfInsert (v, c);
}
```

Last command facilities

The keymap package provides facilities for keeping track of the last command that the user has entered. This information is useful if the behavior of procedures needs to be depend upon previous procedures that the user has called. For example, the *Document* package (see p. 151) makes the behavior of *ctrl-n* and *ctrl-p* depend upon whether the previous character was a *ctrl-n* or *ctrl-p*.

To work with these facilities, you must allocate a last command number for each state that you need to keep track of. Then, when the state occurs, you should call LCSet with the appropriate last command number as an argument. Finally, when you need to test whether the last command was one of the states you are interested in, you should call LCGet to get the value of the last command.

Generating a new last command number

```
long keymap_LCAlloc()
```

Description. Generates a new last command flag. The numbers are allocated by shifting, so they can be AND-ed with the value of *keymap_LCGet*. Generates a maximum of 32 unique numbers. When it reaches 32, begins at 1 again.

Return value. Returns the newly allocated number.

Usage. You should allocate a unique last command number for each procedure or set of procedures that needs one. For example, the Document package allocates two: one for keeping track of whether the previous command was a ctrl-k, and another for keeping track of whether the previous command was a ctrl-p or ctrl-n.

Example. `LCMove = keymap_LCAlloc ();`

Setting the last command

```
keymap_LCSet(LCvalue)
long LCvalue;
```

Description. Sets the value of the this command to LCvalue, which should be a number that was allocated from LCAlloc.

Usage. If a command procedure needs to record that it was the last command, it should call LCSet with its LCvalue. For example, *vmds_nextline* calls LCSet.

Example.

```
vcmds_nextline (v)
     register struct view *v; {

     int pos;

     if (keymap_LCGet () & LCMove)
              pos = PreviousMovePosition;
     else  PreviousMovePosition = pos = v->currentposition;
   ...
     keymap_LCSet (LCMove);
}
```

Getting the last command

```
keymap_LCGet()
```

Description. *keymap_LCGet* returns the value of the last command. Commands that have not set a value through the *LCAlloc* and *LCSet* facilities return the value NULL.

Usage. Use *LCGet* in any procedure that needs to know the value of the last command. You will typically want to AND it with an allocated value to see whether the last command was a command that your procedure is interested in.

Example.

```
vcmds_nextline (v)
     register struct view *v; {

     int pos;

     if (keymap_LCGet () & LCMove)
             pos = PreviousMovePosition;
     else  PreviousMovePosition = pos = v->currentposition;
   ...
     keymap_LCSet (LCMove);
       }
```