Events.mod

COLLABORATORS

	TITLE : Events.mod	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		August 30, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Events.mod

1.1 Events

```
MODULE Events;
(*
     $RCSfile: Events.mod $
  Description: Implements classes for managing events

   Created by: fjc (Frank Copeland)
    $Revision: 1.7 $
      $Author: fjc $
        $Date: 1994/09/03 16:19:49 $

  Copyright © 1994, Frank Copeland.
  This file is part of the Oberon-A Library.
  See Oberon-A.doc for conditions of use and distribution.
  ____________________________________________________________________________

  Module Events defines and implements four classes designed to abstract
  and simplify event handling for Amiga programs.

   Overview   Overview of Amiga events and the event handling classes

  Class descriptions

   Signal        Description of the Signal Class
   MessagePort   Description of the MessagePort Class
   IdcmpPort     Description of the IdcmpPort Class
   EventLoop     Description of the EventLoop Class

  Other

   Switches         Global compiler switches
   Imports          Imported modules
   Initialisation   Module initialisation
   Terminology      Definition of terms
   History          Revision history
```

1.2 Overview of Amiga events and the event handling classes

[It is assumed that you are familiar with the Amiga event handling system, specifically Exec Signals and MsgPorts and Intuition IDCMP ports.]

The Amiga event handling system consists, like most of the system software, of objects and concepts which are built up in layers. At the lowest level, Signals are used to notify Tasks of events. MsgPorts are built on top of Signals to provide message-based event handling. The Intuition IDCMP system builds on the MsgPort system by defining a specific format for messages and a set of standard event types.

Any event-based Amiga program must contain an inner event loop, where it waits for Signals and responds to the ones it receives. In most cases these Signals will be associated with one or more MsgPorts, so the loop must also contain code to remove Messages from the MsgPorts and deal with them. Most MsgPorts will be associated with Intuition Windows, and so the event loop must include code to identify and deal with the IntuiMessages it receives.

This common behaviour presents an opportunity for the creation of classes that can be re-used by any number of programs. This module defines three classes (Signal, MessagePort and IdcmpPort) that abstract the Exec Signal and MsgPort mechanisms, and the Intuition IDCMP mechanism. A programmer will typically extend one or more of these classes to implement the specific behaviour required by an application. A fourth class (EventLoop) abstracts the event loop itself. These four classes can be used as the basis for the event handling of any event-based Amiga application. There is also scope for the creation of other general classes to deal with events generated by, for instance, the Timer device and ARexx.

1.3 Revision History

$Revision: 1.7 $
$Date: 1994/09/03 16:19:49 $

$Log: Events.mod $
Revision 1.7 1994/09/03 16:19:49 fjc
*** empty log message ***

Revision 1.6 1994/08/08 16:19:17 fjc
Release 1.4

Revision 1.5 1994/06/14 02:06:07 fjc
- Updated for release

Revision 1.4 1994/06/09 14:18:21 fjc
- Incorporated changes to Amiga interface

Revision 1.3 1994/06/04 15:50:17 fjc
- Changed to use new Amiga interface

```
  Revision 1.2  1994/05/19  23:49:20  fjc
  - Added OBERON-A: path to links

  Revision 1.1  1994/05/12  19:55:20  fjc
  - Prepared for release
```

1.4 Global Switches

```
  $C= CaseChk       $I= IndexChk  $L+ LongAdr   $N- NilChk
  $P- PortableCode  $R= RangeChk  $S= StackChk  $T= TypeChk
  $V= OvflChk       $Z= ZeroVars

  NIL checking is disabled, and procedures make explicit checks for NIL
  pointers using ASSERT().
```

1.5 Imported Modules

```
IMPORT
  E := Exec, EU := ExecUtil, I := Intuition, SYS := SYSTEM;

(*
  Types declares basic types used by most Amiga libraries.  Exec is the
  central kernel of the Amiga OS.  ExecUtil provides support procedures
  associated with the Exec library.  Intuition provides the Amiga's
  user interface.
```

1.6 Description of Signal

```
  The Signal class is an abstraction of the Exec library's Signal
  mechanism.  A Signal object has one field, sigBit, which contains the
  bit number of the Exec Signal it corresponds to.  Its behaviour is
  defined in two methods, SimpleLoop() and HandleSig().  SimpleLoop()
  implements a simple event loop which waits for the object's Signal to be
  received by the Task.  HandleSig() contains the code that is to be
  executed when the object's Signal is received.
  ____________________________________________________________________________

   Declarations   Constant and type declarations
   Methods        Signal methods
   Usage          Using Signal objects
```

1.7 Using Signal

Signal is an abstract class and Signal objects perform no useful work. To make use of it, the programmer must create a concrete class by extending Signal and implementing the desired behaviour. At a minimum, the subclass must override the HandleSig method and substitute a method which performs whatever action is triggered by the receipt of the signal associated with the object.

The minimum initialisation required for a Signal object is to set the 'sigBit' field to the value of the Exec Signal associated with the object. For a discussion of the minimum behaviour expected of a Signal object, see HandleSig().

As Exec Signals are global to the Task, only one Signal object per Exec Signal can be active within a Task at any one time.

An event loop involving a single Signal object, and hence a single Signal, is implemented in the SimpleLoop procedure. If an event loop involving several Signal objects is required, the programmer should create an EventLoop object.

Example An example of a class derived from Signal

1.8 An example of a class derived from Signal

```
(*
  A Break object is associated with one of the break signals
  defined in Dos.mod.  When one of these signals is received, the
  program is expected to abort.  This class is not very useful, as the
  only input handler that routinely reports these signals is the console
  handler. This is normally only associated with CLI programs, which do
  not use an event loop.
*)

TYPE
  Break = POINTER TO BreakRec;
  BreakRec = RECORD (SignalRec) END;

PROCEDURE (b : Break) HandleSig * () : INTEGER;
(* Overrides the method defined by Signal *)
BEGIN
  RETURN StopAll (* Stop the event loop and exit the program *)
END HandleSig;
...
VAR breakC : Break;
...
BEGIN
  ...
  NEW (breakC);
  breakC.sigBit := Dos.sigBreakCtrlC;
  ...
  SimpleLoop (breakC);
  ...
  (* Clean up and exit *)
```

```
  ...
END ...
```

1.9 Description of MessagePort

The MessagePort class is an extension of the Signal class that abstracts the Exec MsgPort mechanism. A MessagePort object is associated with an Exec MsgPort and its Signal. It has one new field, port, which is a pointer to a MsgPort structure. The MessagePort class overrides the Signal class HandleSig() method and replaces it with a version which removes and replies to any Messages queued at the object's MsgPort. It defines a new method, HandleMsg(), which is responsible for dealing with individual Messages. Other new methods deal with the creation and management of MsgPorts.

Declarations	Constant and type declarations
Methods	MessagePort methods
Usage	Using MessagePort objects

1.10 Using MessagePort

Like Signal, MessagePort is an abstract class and must be extended in order to be useful. The derived class must at the least override the HandleMsg() method and replace it with an implementation of the desired behaviour.

A MessagePort object must be initialised with either the AttachPort() or the MakePort() method. AttachPort() is used when the programmer wishes to associate the object with a pre-existing MsgPort, such as one belonging to an Intuition window. MakePort() is used to create an entirely new MsgPort. Both these methods initialise the object's 'sigBit' field. If AttachPort() is used, it is up to the programmer to ensure that only one MessagePort object is associated with that MsgPort at any one time.

A MessagePort object must be cleaned up when it is no longer required. If it was initialised with AttachPort(), call DetachPort(). Similarly, call DeletePort() to clean up an object initialised with MakePort().

FlushPort() is mainly used by other methods when detaching Exec MsgPorts from MessagePort objects. However, under some circumstances user programs may need to use it directly.

Example	An example of using MessagePort.

1.11 Description of IdcmpPort

The IdcmpPort class extends the MessagePort class to deal with IntuiMessages from an IDCMP message port. It overrides the HandleMsg() method with an implementation which passes the IntuiMessage to a handler procedure depending on the value in the Class field. The programmer must declare and install handler procedures for each class of IntuiMessage expected by the program.

1.12 Using IdcmpPorts

Unlike the Signal and MessagePort classes, IdcmpPort is a concrete class that may be used directly. Extensions of IdcmpPort will typically override the SetupWindow() and CleanupWindow() methods to add menus and gadgets to the Window and to remove them.

The Init() method must be called before an IdcmpPort object is used to ensure that it is in a safe state. The programmer must also install at least one handler procedure in the object (See DefaultHandler() for a description of the behaviour required of handler procedures).

Like any MessagePort object, an IdcmpPort object must be associated with an Exec MsgPort, specifically the UserPort field of an open Intuition Window. If the programmer allows Intuition to create this MsgPort, (s)he should use AttachPort() to connect the object to it after the Window is opened. If the programmer wishes to use one MsgPort for several windows, (s)he may use MakePort() to create it or AttachPort() to connect to one created elsewhere. In this case the programmer must make sure the handler procedures installed in the object are able to determine which window is active (this can be found in the IntuiMessage).

The final step is to call the SetupWindow() method to make any modifications to the Window that the IdcmpPort requires. This could involve adding Menus and Gadgets to it, for example.

The IdcmpPort object can now be placed in its own event loop using the SimpleLoop() method, or attached to an EventLoop object to share an event loop with other Signal objects.

When the IdcmpPort object is no longer required, clean up the Window with the CleanupWindow() method. Then use the DetachPort() or DeletePort() method as appropriate.

1.13 An example of using an IdcmpPort object

```
(*-----------------------------------*)
PROCEDURE* HandleCloseWindow
  ( ip : IdcmpPort;
    message : Intuition.IntuiMessagePtr )
  : INTEGER;

BEGIN (* HandleCloseWindow *)
  E.base.ReplyMsg (message);
  RETURN Stop
END HandleCloseWindow;

(*-----------------------------------*)
PROCEDURE* HandleGadgetUp
  ( ip : IdcmpPort;
    message : Intuition.IntuiMessagePtr )
  : INTEGER;

  VAR result, gadgetId : INTEGER;

BEGIN (* HandleGadgetUp *)
  result := Pass; gadgetId := message.Code;
  CASE gadgetId OF
    ...
    process gadget releases
    ...
  ELSE
    ...
    default actions
    ...
  END; (* CASE gadgetId *)
  RETURN result
END HandleGadgetUp;
...
VAR
  ip : IdcmpPort;
  nw : Intuition.NewWindow;
  window : Intuition.WindowPtr;
...
BEGIN
  ...
  NEW (ip);
  ip.Init;
  ip.Handle [Intuition.idcmpCloseWindow] := HandleCloseWindow;
  ip.Handle [Intuition.idcmpGadgetUp] := HandleGadgetUp;
  ...
  window := Intuition.base.OpenWindow (nw);
  ...
  AttachPort (ip, window.UserPort);
  ip.SetupWindow (window);
  ...
  SimpleLoop (ip);
  ...
  ip.CleanupWindow (window);
  DetachPort (ip, window.UserPort);
  ...
  Intuition.base.CloseWindow (window);
```

```
  ...
END
...
```

1.14 Description of EventLoop

A EventLoop object is used to group a number of Signal objects that are to be processed together using a single event loop. The associated Signal objects must be created and initialised by the programmer before adding them to the EventLoop object.

Declarations	Constant and type declarations
Methods	EventLoop methods
Usage	Using EventLoop objects

1.15 Using EventLoop

The EventLoop class is a concrete class that is intended to be used directly. A EventLoop object is used to group together several Signal objects and process them all in a single event loop.

InitEventLoop() must be called to initialise an EventLoop object before it can be used. One or more calls to AddSignal() should then be made to attach Signal objects to the EventLoop object.

The Loop () method implements an event loop using all the Signal objects attached to the EventLoop object. At least one Signal object should be attached to the EventLoop object before calling it. Further Signal objects may be attached after the call, but only Signal objects already attached. Signal objects are automatically removed from the EventLoop when their HandleSig() methods return Stop. The method exits when all Signal objects have been removed or when one object's HandleSig() method returns StopAll.

The RemoveSignal() method will not normally be used, as individual Signal objects know best when they should be de-activated.

Example An example of using an EventLoop object.

1.16 An example of using an EventLoop object

```
...
VAR
  sig, ignore : Signal;
  mp : MessagePort;
  ip : IdcmpPort;
  eventLoop : EventLoop;
...
```

```
  BEGIN
    ...
    NEW (eventLoop); InitEventLoop (eventLoop);
    ...
    NEW (sig); NEW (mp); NEW (ip);
    ...
    ignore := AddSignal (eventLoop, sig);
    ignore := AddSignal (eventLoop, mp);
    ignore := AddSignal (eventLoop, ip);
    ...
    Loop (eventLoop);
    ...
    Clean up
    ...
  END
```

1.17 Signal Declarations

```
(*
  Signal object fields:

  sigBit -- the number of the Exec signal the object handles.  This must
  be a positive integer < 32 or -1 (meaning no signal).
*)

TYPE

  Signal *= POINTER TO SignalRec;
  SignalRec *= RECORD
    sigBit *: SHORTINT;
  END; (* SignalRec *)

(*
  These are the valid return codes for the HandleSig method.
*)

CONST
  Pass     *= 0;
    (* Did not handle the event. *)
  Continue *= 1;
    (* Finished handling this event, wait for the next. *)
  Stop     *= 2;
    (* Stop processing events for this handler. *)
  StopAll  *= 3;
    (* Stop processing events for all handlers; halt the Task *)

(*
```

1.18 MessagePort Declarations

```
(*
  MessagePort object fields:
```

```
  port -- the MsgPort the object gets messages from.  This is read-only
  and is modified through the AttachPort(), DetachPort(), MakePort(), and
  DeletePort() methods.
*)

TYPE

  MessagePort *= POINTER TO MessagePortRec;
  MessagePortRec *= RECORD (SignalRec)
    port -: E.MsgPortPtr;
  END; (* MessagePortRec *)

(*
```

1.19 IdcmpPort Declarations

```
(*
  IdcmpPort object fields:

  Handle -- the table of handler procedures, one for each possible class
  of IntuiMessage.  The Init() methods assigns DefaultHandler to each.
  The procedure must return one of the return codes listed for HandleSig
  procedures (see Signal).  A do-nothing procedure simply returns 'Pass'.
*)

TYPE

  IdcmpPort *= POINTER TO IdcmpPortRec;
  IdcmpProc *=
    PROCEDURE ( ip : IdcmpPort; msg : I.IntuiMessagePtr ) : INTEGER;

  IdcmpPortRec * = RECORD (MessagePortRec)
    Handle *: ARRAY 32 OF IdcmpProc;
  END; (* IdcmpPortRec *)

(*
```

1.20 Data Structures

```
CONST
  NumSignals = 32; (* The maximum number of signals for a Task. *)

(*
  EventLoop object fields:

  sigBits -- the combined signal bits of all the Signal objects attached
  to the EventLoop.

  signal -- the table of Signal objects attached to the EventLoop.
*)
```

```
TYPE

  EventLoop *= POINTER TO EventLoopRec;
  EventLoopRec *= RECORD
    sigBits : SET;
    signal  : ARRAY NumSignals OF Signal;
  END; (* EventLoopRec *)

(*
```

1.21 Signal Methods

```
  NEW METHODS

  Event handling

    PROCEDURE ( signal : Signal ) HandleSig * () : INTEGER;
    - Implements the behaviour associated with a given Signal.

  Event Loop

    PROCEDURE SimpleLoop * ( signal );
    - Executes a simple event loop.
```

1.22 HandleSig()

```
PROCEDURE (h : Signal) HandleSig * () : INTEGER;
(*
  The HandleSig method implements the primary behaviour of the Signal
  class and its descendants.  It is called by the SimpleLoop() method (or
  the Loop() method of an EventLoop object) when the sigBit associated with
  the object is received by the Task.

  Each descendant class is expected to override this method and provide
  its own implementation.  In order for the SimpleLoop (and Loop) methods to
  work, the replacement methods must implement at least the following
  behaviour:

  - If the method performs no action for a given event, it must return the
    constant 'Pass'.
  - If the Signal no longer needs to be part of the event loop (ie- when a
    Window is closed), it must return 'Stop'.
  - If the event causes the program to terminate, the method must return
    'StopAll'.
  - In all other cases it must return 'Continue'.
*)

BEGIN (* HandleSig *)
  HALT (99); (* Abort the program, method not implemented. *)
  RETURN StopAll (* This is superfluous *)
END HandleSig;
(*
```

1.23 SimpleLoop()

```
PROCEDURE SimpleLoop * ( sig : Signal );
(*
  Implements an event loop which waits for a single Exec Signal.  The
  Signal object must be fully initialised before calling this method.
*)

  VAR signalsReceived : SET; result : INTEGER;

BEGIN (* SimpleLoop *)
  ASSERT (sig # NIL, 132);
  REPEAT
    signalsReceived := E.base.Wait ({sig.sigBit});
    result := sig.HandleSig ()
  UNTIL (result > Continue);
END SimpleLoop;
(*
```

1.24 MessagePort Methods

```
  OVERRIDDEN METHODS

  Event handling

    PROCEDURE (VAR mp : MessagePort) HandleSig * () : INTEGER;
    - Removes and processes messages queued at the object's MessagePort.

  NEW METHODS

  Initialisation

    PROCEDURE MakePort* ( mp, name, priority ) : BOOLEAN;
    - Creates a MessagePort and attaches it to the object.

    PROCEDURE DeletePort* ( mp );
    - Detaches the object's MessagePort and disposes of it.

    PROCEDURE AttachPort* ( mp, port );
    - Attaches an existing MessagePort to the object.

    PROCEDURE DetachPort* ( mp );
    - Detaches the object's MessagePort.

  Message Port maintenance

    PROCEDURE ( mp : MessagePort ) FlushPort* ();
    - Flushes any pending messages from the object's MessagePort.

  Event handling

    PROCEDURE ( mp : MessagePort ) HandleMsg* ( msg ) : INTEGER;
    - Performs the action required for a given message.
```

1.25 HandleSig()

```
PROCEDURE^ (mp : MessagePort) HandleMsg * ( msg : E.MessagePtr ) : INTEGER;

PROCEDURE (mp : MessagePort) HandleSig * () : INTEGER;
(*
  This procedure implements the behaviour required by receiving the signal
  associated with a MsgPort.  The actual handling of the messages is
  delegated to mp.HandleMsg().
*)

  VAR result : INTEGER; msg : E.MessagePtr;

BEGIN (* HandleSig *)
  result := Pass; (* Default *)

  (* Loop until all messages queued at the port are dealt with *)
  LOOP
    (* Dequeue the next message, quit if there is none *)
    msg := E.base.GetMsg (mp.port);
    IF msg = NIL THEN EXIT END;

    (* Despatch to the message handler *)
    result := mp.HandleMsg (msg);

    (* Process the return code *)
    IF result = Pass THEN E.base.ReplyMsg (msg) END;
    IF result > Continue THEN EXIT END
  END;
  RETURN result
END HandleSig;
(*
```

1.26 HandleMsg()

```
PROCEDURE (mp : MessagePort) HandleMsg * ( msg : E.MessagePtr ) : INTEGER;

(*
  This method is responsible for dealing with individual Messages after
  they have been remove from the MsgPort queue with E.base.GetMsg().  Each
  descendant class is expected to override this method and provide its own
  implementation.  In order for the SimpleLoop (and Loop) methods to work,
  the replacement methods must implement at least the following behaviour:

  - If the method performs no action for a given Message, it must return
    the constant 'Pass'.
  - If it is no longer necessary to wait for Messages at the MsgPort, the
    method should return 'Stop'.
  - If the Message causes the program to terminate, the method must return
    'StopAll'.
  - In all other cases it must return 'Continue'.

  If the method returns any result other than 'Pass' it must first call
  'E.base.ReplyMsg (msg)' to remove the Message from the MsgPort.  If
```

```
  it returns 'Pass', it should *never* call ReplyMsg().
*)

BEGIN (* HandleMsg *)
  HALT (99); (* Abort the program, method not implemented. *)
  RETURN StopAll (* This is superfluous *)
END HandleMsg;
(*
```

1.27 FlushPort()

```
PROCEDURE (mp : MessagePort) FlushPort * ();
(*
  Gets and replies to all messages queued for the handler's message
  port.  It is called inside an Exec.Forbid()/Exec.Permit() pair to
  prevent more messages from arriving at the port while it is being
  flushed.
*)

  VAR msg : E.MessagePtr;

BEGIN (* FlushPort *)
  E.base.Forbid ();
    LOOP
      msg := E.base.GetMsg (mp.port);
      IF msg = NIL THEN EXIT END;
      E.base.ReplyMsg (msg)
    END;
  E.base.Permit ()
END FlushPort;
(*
```

1.28 AttachPort()

```
PROCEDURE AttachPort* ( mp : MessagePort; port : E.MsgPortPtr );
(*
  Attaches an Exec message port to a handler.
*)

BEGIN (* AttachPort *)
  ASSERT (mp # NIL, 132);
  ASSERT (port # NIL, 132);
  mp.sigBit := port.sigBit;
  mp.port := port;
END AttachPort;
(*
```

1.29 DetachPort()

```
PROCEDURE DetachPort *
   ( mp : MessagePort );
(*
  Detaches a message port from a handler after flushing any remaining
  messages.
*)

BEGIN (* DetachPort *)
  ASSERT (mp # NIL, 132);
  mp.FlushPort ();
  mp.port := NIL;
  mp.sigBit := -1;
END DetachPort;
(*
```

1.30 MakePort()

```
PROCEDURE MakePort *
   ( mp : MessagePort; name : ARRAY OF CHAR; priority : SHORTINT )
   : BOOLEAN;
(*
  Creates an Exec message port and attaches it to the handler.
*)

  VAR port : E.MsgPortPtr;

BEGIN (* MakePort *)
  ASSERT (mp # NIL, 132);
  port := EU.CreatePort (name, priority);
  IF port # NIL THEN AttachPort (mp, port); RETURN TRUE
  ELSE RETURN FALSE
  END
END MakePort;
(*
```

1.31 DeletePort()

```
PROCEDURE DeletePort * ( mp : MessagePort );
(*
  Deletes a message port created with MakePort() and frees any resources
  allocated to it.
*)

BEGIN (* DeletePort *)
  ASSERT (mp # NIL, 132);
  E.base.Forbid ();
    mp.FlushPort ();
    EU.DeletePort (mp.port);
  E.base.Permit ();
  mp.port := NIL;
  mp.sigBit := -1
```

```
END DeletePort;
(*
```

1.32 DefaultHandler()

```
PROCEDURE* DefaultHandler
  ( ip : IdcmpPort; msg : I.IntuiMessagePtr )
  : INTEGER;
(*
  The default handler for IntuiMessage classes that do not have a
  specific handler provided.
*)

BEGIN (* DefaultHandler *)
  RETURN Pass
END DefaultHandler;
(*
```

1.33 IdcmpPort Methods

```
  OVERRIDDEN METHODS

  Event handling

    PROCEDURE (ip : IdcmpPort) HandleMsg* ( msg ) : INTEGER;
    - Performs the action required for an IntuiMessage

  NEW METHODS

  Initialisation

    PROCEDURE (ip : IdcmpPort) Init* ();
    - Initialises an IdcmpPort object.

    PROCEDURE (ip : IdcmpPort) SetupWindow* ( window );
    - Associates a handler object with an Intuition window.

    PROCEDURE (ip : IdcmpPort) CleanupWindow* ( window );
    - Dissociates a handler object from an Intuition window.

  PRIVATE METHODS

    PROCEDURE DefaultHandler ( ip, message ) : INTEGER;
    - Default handler for IDCMP events: does nothing.
```

1.34 HandleMsg()

```
PROCEDURE (ip : IdcmpPort) HandleMsg* ( msg : E.MessagePtr ) : INTEGER;
(*
  This implementation assumes that any message received at an Intuiton
```

```
  Window's MsgPort is an IntuiMessage, an extension of an Exec
  Message.  Each IntuiMessage has a Class field, which indicates which
  kind of event the IntuiMessage is reporting.  The actual handling of the
  message is delegated to one of the ip.HandleIdcmp() procedures,
  depending on the value in the Class field.

  The event type is indicated by the setting of a single bit in the Class
  field. This gives a maximum of 32 possible event types, not all of which
  are currently defined.  The actual event is determined by searching
  through the bits of the Class field until a set bit is found.
*)

  VAR
    intuiMessage : I.IntuiMessagePtr;
    class : SET; flag : SHORTINT;

BEGIN (* HandleMsg *)
  (* Type cast message to an IntuiMessagePtr *)
  intuiMessage := SYS.VAL (I.IntuiMessagePtr, msg);

  (* Get the Class field *)
  class := intuiMessage.class;

  (* Search for the event type and despatch to the appropriate handler *)
  FOR flag := 0 TO 31 DO
    IF flag IN class THEN
      (* $N+ Turn on NIL checking just for the call *)
      RETURN ip.Handle [flag] (ip, intuiMessage)
      (* $N- *)
    END
  END;

  RETURN Pass (* Default if no bit set in class *)
END HandleMsg;
(*
```

1.35 Init()

```
PROCEDURE (ip : IdcmpPort) Init* ();
(*
  The initialisation task is to assign DefaultHandler to all of the
  handlers in the handler table.  This ensures that the receipt of an
  unexpected message doesn't ruin your day.
*)

  VAR flag : SHORTINT;

BEGIN (* Init *)
  FOR flag := 0 TO 31 DO ip.Handle [flag] := DefaultHandler END
END Init;
(*
```

1.36 SetupWindow()

```
PROCEDURE (ip : IdcmpPort) SetupWindow* (window : I.WindowPtr);
(*
  This method is intended to be overridden by extensions of IdcmpPort
  to take care of any processing required after the window is attached to
  the handler but before it starts to process messages.  One possible use
  is for attaching Menus and Gadgets to the window.
*)

BEGIN (* SetupWindow *)
END SetupWindow;
(*
```

1.37 CleanupWindow()

```
PROCEDURE (ip : IdcmpPort) CleanupWindow* (window : I.WindowPtr);
(*
  This method is intended to be overridden by extensions of IdcmpPort
  to take care of any processing required after the window is detached
  from the object.  One possible use is for removing Menus from the
  window.
*)

BEGIN (* CleanupWindow *)
END CleanupWindow;
(*
```

1.38 EventLoop Methods

```
  NEW METHODS

  Initialisation

    PROCEDURE InitEventLoop* ( el );
    - Initialises an EventLoop object.

    PROCEDURE AddSignal* ( el, signal ) : Signal;
    - Add a Signal object to a EventLoop object

    PROCEDURE RemoveSignal* ( el, signal );
    - Removes a Signal object from a EventLoop object.

  Event Handling

    PROCEDURE Loop* ( el );
    - Starts the event loop.
```

1.39 InitEventLoop()

```
PROCEDURE InitEventLoop* ( el : EventLoop );
(*
  Performs necessary initialisation for an EventLoop object.  It simply
  zeroes all fields.
*)

  VAR index : INTEGER;

BEGIN (* InitEventLoop *)
  ASSERT (el # NIL, 132);
  el.sigBits := {};
  FOR index := 0 TO NumSignals - 1 DO
    el.signal [index] := NIL
  END;
END InitEventLoop;
(*
```

1.40 AddSignal()

```
PROCEDURE AddSignal* ( el : EventLoop; signal : Signal ) : Signal;
(*
  Adds a Signal object to an EventLoop object.  The Signal must be fully
  initialised first.  If there is already a Signal for the same sigBit it
  is returned; this allows Signals to be temporarily replaced and restored
  with successive calls.
*)

  VAR sigBit : SHORTINT; oldSignal : Signal;

BEGIN (* AddSignal *)
  ASSERT (el # NIL, 132);
  ASSERT (signal # NIL, 132);
  sigBit := signal.sigBit;
  oldSignal := el.signal [sigBit];
  INCL (el.sigBits, sigBit);
  el.signal [sigBit] := signal;
  RETURN oldSignal
END AddSignal;
(*
```

1.41 RemoveSignal()

```
PROCEDURE RemoveSignal* ( el : EventLoop; signal : Signal );
(*
  Removes a Signal object from an EventLoop object.
*)

  VAR sigBit : SHORTINT;

BEGIN (* RemoveSignal *)
  ASSERT (el # NIL, 132);
```

```
  ASSERT (signal # NIL, 132);
  sigBit := signal.sigBit;
  IF el.signal [sigBit] = signal THEN
    el.signal [sigBit] := NIL;
    EXCL (el.sigBits, sigBit);
  END
END RemoveSignal;
(*
```

1.42 Loop()

```
PROCEDURE Loop* ( el : EventLoop );
(*
  Implements an event loop using multiple Signal objects.  The loop
  continues until all Signal objects have returned Stop or one has
  returned StopAll.
*)

  VAR
    signalsReceived : SET; sigBit : SHORTINT; result : INTEGER;
    signal : Signal;

BEGIN (* Loop *)
  ASSERT (el # NIL, 132);
  (* Loop while there are active Signals *)
  WHILE el.sigBits # {} DO

    (* Wait for signal(s) to arrive *)
    signalsReceived := E.base.Wait (el.sigBits);

    (* For each signal received... *)
    FOR sigBit := 0 TO NumSignals - 1 DO
      IF sigBit IN signalsReceived THEN

        (* Get the relevant Signal *)
        signal := el.signal [sigBit];
        ASSERT (signal # NIL, 132);

        (* Handle the signal *)
        result := signal.HandleSig ();

        IF result = Stop THEN

          (* Remove the Signal *)
          el.signal [sigBit] := NIL;
          EXCL (el.sigBits, sigBit)

        ELSIF result = StopAll THEN

          (* Exit the event loop *)
          el.sigBits := {}
        END
      END (* IF *)
    END (* FOR *)
  END (* WHILE *)
```

```
END Loop;
(*
```

1.43 Module Initialisation

```
(* No initialisation required *)

END Events.
(*
```