00c76cd0-11

COLLABORATORS

	TITLE : 00c76cd0-11	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		December 7, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

00c76cd0-11

1.1 "

```
   $RCSfile: Events.doc $
Description: Documentation for event handling classes.

 Created by: fjc (Frank Copeland)
  $Revision: 1.3 $
    $Author: fjc $
      $Date: 1995/06/29 18:56:58 $
```

```
Module Events exports five classes that abstract the Amiga event
handling system. Four of the classes implement object-oriented shells
around the basic system components: Exec Signals and MsgPorts, Intuition
IDCMP ports and GadTools IDCMP ports. The remaining class abstracts the
standard event loop.

~Overview~~~~~  Overview of Amiga event handling
~Definition~~~  Definition of module Events
~Source~~~~~~~  Source code of module Events
~Terminology~~  Explaining the jargon.

Classes

~Signal~~~~~~~  Abstraction of Exec Signals
~MessagePort~~  Abstraction of Exec MsgPorts
~IdcmpPort~~~~  Abstraction of Intuition IDCMP ports
~GadToolsPort~  Abstraction of GadTools IDCMP ports
~EventLoop~~~~  Abstraction of the event loop
```

1.2 The Signal class -- abstracting Exec Signals

```
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
contained 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.

~Definition~ Definition of the Signal class
~Usage~~~~~~ Using the Signal class
~Example~~~~ Signal class example

1.3 The MessagePort class -- abstracting Exec MsgPorts

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.

~Definition~ Definition of the MessagePort class
~Usage~~~~~~ Using the MessagePort class
~Example~~~~ MessagePort class example

1.4 The IdcmpPort class -- abstracting Intuition IDCMP ports

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.

~Definition~ Definition of the IdcmpPort class
~Usage~~~~~~ Using the IdcmpPort class
~Example~~~~ IdcmpPort class example

1.5 The GadToolsPort class -- abstracting GadTools IDCMP ports

GadToolsPort is an extension of the IdcmpPort class that is intended to be used with the high-level user interface objects provided by the GadTools library. It is identical in interface and operation to the IdcmpPort class, except that it uses the GadTools library functions for obtaining and replying to messages from a Window's IDCMP port.

Simply use an extension of GadToolsPort in place of an extension of IdcmpPort whenever a Window contains GadTools gadgets.

1.6 The EventLoop class -- abstracting the event loop

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.

~Definition~ Definition of the EventLoop class
~Usage~~~~~~ Using the EventLoop class
~Example~~~~ EventLoop class example

1.7 Overview of the Amiga event handling system

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. The GadTools IDCMP system extends the Intuition system to support the handling of higher-level user interface objects.

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 four classes (Signal, MessagePort, IdcmpPort and GadToolsPort) 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 fifth class (EventLoop) abstracts the event loop itself. These five 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.

1.8 Definition of the Signal class

```
TYPE

  Signal * = POINTER TO SignalRec;
```

```
  SignalRec * = RECORD
    sigBit : SHORTINT; (* The signal bit to wait for *)

    PROCEDURE (h : Signal) HandleSig () : INTEGER;
    -- Performs the appropriate action when the signal is received.
  END;

CONST

  (* These are the legal return codes from Signal.HandleSig() *)

  Pass     = 0;  (* Did not handle the signal                 *)
  Continue = 1;  (* Handled the signal, continue running      *)
  Stop     = 2;  (* Handled the signal, stop listening for it *)
  StopAll  = 3;  (* Handled the signal, exit the event loop   *)

  (* Disables garbage collection *)

  NoGC = 0;

PROCEDURE SimpleLoop ( sig : Signal; collectFreq : INTEGER );
-- Implements a simple event loop which listens for the Exec Signal
   associated with the 'sig' parameter. The 'collectFreq' parameter
   determines how frequently the garbage collector is called. Pass
   NoGC if you don't want it called at all.
```

1.9 Using the Signal class

Signal is an abstract class and Signal objects perform no useful work. To make use of the class, 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. 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.

The HandleSig method implements the primary behaviour of the Signal class and its descendants. It is called by the SimpleLoop() procedure (or the Do() 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 EventLoop.Do) 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'.

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.

The collectFreq parameter of SimpleLoop determines how often the
garbage collector will be called. The value passed is the number of
times the signal must be received before the garbage collector is
activated. The value passed depends on the application, and how often
the signal is likely to be received. For example, a window receiving
intuiTick events will get approximately 6 per second, so a collectFreq
value of 60 will activate the garbage collector approximately every 10
seconds. A window receiving mouseMove events will be notified *many*
times per second, and a correspondingly higher value must be used. If
you don't wish to use the garbage collector, pass the constant 'NoGC'.

If any value other than NoGC is used for collectFreq, care must be
taken to make sure that there are no 'live' local pointer variables
when SimpleLoop is called. See the section on Garbage~Collection in
OC.doc.
```

1.10 Signal class example

```
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.ctrlC;
  ...
  SimpleLoop (breakC, NoGC);
  ...
  (* Clean up and exit *)
  ...
```

```
END ...
```

1.11 Definition of the MessagePort class

```
TYPE

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

    PROCEDURE (mp : MessagePort) HandleMsg
      ( msg : Exec.MessagePtr )
      : INTEGER;
    -- Handles the receipt of a message at the object's MsgPort.

    PROCEDURE (mp : MessagePort) FlushPort;
    -- Flushes all pending messages at the object's MsgPort.

    PROCEDURE (mp : MessagePort) AttachPort
      ( port : Exec.MsgPortPtr );
    -- Attaches an existing MsgPort to the object.

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

    PROCEDURE (mp : MessagePort) MakePort
      ( name : ARRAY OF CHAR;
        priority : SHORTINT )
      : BOOLEAN;
    -- Creates a MsgPort and attaches it to the object.

    PROCEDURE (mp : MessagePort) DeletePort;
    -- Deletes the MsgPort attached to the object.

  END;
```

1.12 Using the MessagePort class

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.

The HandleMsg() method is responsible for dealing with individual
Exec Messages after they have been remove from the MsgPort queue with
Exec.GetMsg().  Each descendant class is expected to override this
method and provide its own implementation.  In order for the SimpleLoop
(and EventLoop.Do) 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
'Exec.ReplyMsg (msg)' to remove the Message from the MsgPort.  If it
returns 'Pass', it should *never* call Exec.ReplyMsg().
```

1.13 MessagePort class example

```
See IdcmpPort.
```

1.14 Definition of the IdcmpPort class

```
TYPE

  IdcmpPort * = POINTER TO IdcmpPortRec;

  IdcmpPortRec * = RECORD (MessagePortRec)
    PROCEDURE (ip : IdcmpPort) SetupWindow
      ( window : Intuition.WindowPtr );
    -- Associates an IdcmpPort object with an Intuition window.

    PROCEDURE (ip : IdcmpPort) CleanupWindow
      ( window : Intuition.WindowPtr );
    -- Breaks the association of an IdcmpPort with an Intuition window.

    PROCEDURE (ip : IdcmpPort) HandleSizeVerify
      ( msg : Intuition.IntuiMessagePtr )
      : INTEGER;
    PROCEDURE (ip : IdcmpPort) HandleNewSize
      ( msg : Intuition.IntuiMessagePtr )
      : INTEGER;
```

```
PROCEDURE (ip : IdcmpPort) HandleRefreshWindow
  ( msg : Intuition.IntuiMessagePtr )
  : INTEGER;
PROCEDURE (ip : IdcmpPort) HandleMouseButtons
  ( msg : Intuition.IntuiMessagePtr )
  : INTEGER;
PROCEDURE (ip : IdcmpPort) HandleMouseMove
  ( msg : Intuition.IntuiMessagePtr )
  : INTEGER;
PROCEDURE (ip : IdcmpPort) HandleGadgetDown
  ( msg : Intuition.IntuiMessagePtr )
  : INTEGER;
PROCEDURE (ip : IdcmpPort) HandleGadgetUp
  ( msg : Intuition.IntuiMessagePtr )
  : INTEGER;
PROCEDURE (ip : IdcmpPort) HandleReqSet
  ( msg : Intuition.IntuiMessagePtr )
  : INTEGER;
PROCEDURE (ip : IdcmpPort) HandleMenuPick
  ( msg : Intuition.IntuiMessagePtr )
  : INTEGER;
PROCEDURE (ip : IdcmpPort) HandleCloseWindow
  ( msg : Intuition.IntuiMessagePtr )
  : INTEGER;
PROCEDURE (ip : IdcmpPort) HandleRawKey
  ( msg : Intuition.IntuiMessagePtr )
  : INTEGER;
PROCEDURE (ip : IdcmpPort) HandleReqVerify
  ( msg : Intuition.IntuiMessagePtr )
  : INTEGER;
PROCEDURE (ip : IdcmpPort) HandleReqClear
  ( msg : Intuition.IntuiMessagePtr )
  : INTEGER;
PROCEDURE (ip : IdcmpPort) HandleMenuVerify
  ( msg : Intuition.IntuiMessagePtr )
  : INTEGER;
PROCEDURE (ip : IdcmpPort) HandleNewPrefs
  ( msg : Intuition.IntuiMessagePtr )
  : INTEGER;
PROCEDURE (ip : IdcmpPort) HandleDiskInserted
  ( msg : Intuition.IntuiMessagePtr )
  : INTEGER;
PROCEDURE (ip : IdcmpPort) HandleDiskRemoved
  ( msg : Intuition.IntuiMessagePtr )
  : INTEGER;
PROCEDURE (ip : IdcmpPort) HandleActiveWindow
  ( msg : Intuition.IntuiMessagePtr )
  : INTEGER;
PROCEDURE (ip : IdcmpPort) HandleInactiveWindow
  ( msg : Intuition.IntuiMessagePtr )
  : INTEGER;
PROCEDURE (ip : IdcmpPort) HandleDeltaMove
  ( msg : Intuition.IntuiMessagePtr )
  : INTEGER;
PROCEDURE (ip : IdcmpPort) HandleVanillaKey
  ( msg : Intuition.IntuiMessagePtr )
  : INTEGER;
```

```
    PROCEDURE (ip : IdcmpPort) HandleIntuiTicks
      ( msg : Intuition.IntuiMessagePtr )
      : INTEGER;
    PROCEDURE (ip : IdcmpPort) HandleIdcmpUpdate
      ( msg : Intuition.IntuiMessagePtr )
      : INTEGER;
    PROCEDURE (ip : IdcmpPort) HandleMenuHelp
      ( msg : Intuition.IntuiMessagePtr )
      : INTEGER;
    PROCEDURE (ip : IdcmpPort) HandleChangeWindow
      ( msg : Intuition.IntuiMessagePtr )
      : INTEGER;
    PROCEDURE (ip : IdcmpPort) HandleGadgetHelp
      ( msg : Intuition.IntuiMessagePtr )
      : INTEGER;
    -- Each of these methods handles a single class of IDCMP message.

    PROCEDURE (ip : IdcmpPort) DefaultHandler
      ( msg : Intuition.IntuiMessagePtr;
        flag : INTEGER )
      : INTEGER;
    -- Default handler for message classes that don't have their own
       handler. Calling this method should be considered a bug, leading
       to the program halting.
  END;
```

1.15 Using the IdcmpPort class

IdcmpPort is an abstract class that must be extended to be useful. Extensions of IdcmpPort will typically override one or more of the Handle[IDCMP class] methods, and may also implement the SetupWindow() and CleanupWindow() methods.

An IdcmpPort object must be initialised using one of the AttachPort() or MakePort() methods, and cleaned up with one of the DetachPort() or DeletePort() methods. When the MsgPort is the userPort of an Intuition Window, use the AttachPort() and DetachPort() methods; it is easiest to make these calls inside the SetupWindow() and CleanupWindow() methods.

The SetupWindow() method can be used to perform other tasks needed to associate an Intuition Window with an IdcmpPort object. These can include creating and attaching Intuition Menus and Gadgets to the window. Removing and deallocating such objects can be performed by the CleanupWindow() method.

Each of the handler procedures for individual IDCMP classes is expected to behave in a similar way to the MessagePort.HandleMsg() method. Each method must implement at least the following behaviour:

- If the method performs no action for a given IntuiMessage, it must return the constant 'Pass'.
- If it is no longer necessary to wait for IntuiMessages at the MsgPort, the method should return 'Stop'.
- If the IntuiMessage 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
'Exec.ReplyMsg (msg)' to remove the IntuiMessage from the MsgPort.  If
it returns 'Pass', it should *never* call Exec.ReplyMsg().

The DefaultHandler() method is intended to trap any messages received
with invalid 'class' fields. It currently causes the program to HALT,
and there is no reason to change this behaviour.
```

1.16 IdcmpPort class example

```
TYPE

  MyPort *= RECORD (Events.IdcmpPort) END;

(*---------------------------------*)
PROCEDURE (p : MyPort) SetupWindow*
  ( window : Intuition.WindowPtr );

BEGIN
  p.AttachPort (window.userPort);
  ...
END SetupWindow;

(*---------------------------------*)
PROCEDURE (p : MyPort) CleanupWindow*
  ( window : Intuition.WindowPtr );

BEGIN
  ...
  p.DetachPort;
END CleanupWindow;

(*---------------------------------*)
PROCEDURE (p : MyPort) HandleCloseWindow*
  ( message : Intuition.IntuiMessagePtr )
  : INTEGER;

BEGIN (* HandleCloseWindow *)
  Exec.ReplyMsg (message);
  RETURN Events.Stop
END HandleCloseWindow;

(*---------------------------------*)
PROCEDURE (p : MyPort) HandleGadgetUp*
  ( message : Intuition.IntuiMessagePtr )
  : INTEGER;

  VAR result, gadgetId : INTEGER;

BEGIN (* HandleGadgetUp *)
  result := Events.Pass; gadgetId := message.Code;
  CASE gadgetId OF
    ...
```

```
    process gadget releases
    ...
  ELSE
    ...
    default actions
    ...
  END; (* CASE gadgetId *)
  RETURN result
END HandleGadgetUp;

...

VAR
  port : MyPort;
  window : Intuition.WindowPtr;
...
BEGIN
  ...
  window := Intuition.OpenWindowTags (...);
  ...
  NEW (port);
  port.SetupWindow (window);
  ...
  SimpleLoop (port, NoGC);
  ...
  port.CleanupWindow (window);
  ...
  Intuition.CloseWindow (window);
  ...
END
...
```

1.17 Definition of the EventLoop class

```
TYPE

  EventLoop * = POINTER TO EventLoopRec;
  EventLoopRec * = RECORD
    PROCEDURE (el : EventLoop) AddSignal
      ( signal : Signal )
      : Signal;
    -- Adds a Signal object to the event loop. If a Signal object with
       the same 'sigBit' value is already installed, it is returned.

    PROCEDURE (el : EventLoop) RemoveSignal
      ( signal : Signal );
    -- Removes a given Signal from the event loop.

    PROCEDURE (el : EventLoop) Collect
      ( collectFreq : INTEGER );
    -- Determines the frequency of garbage collection.

    PROCEDURE (el : EventLoop) Do;
    -- Starts the processing of the event loop. At least one call to
       AddSignal() must be made before this method is called.
```

```
  END;

PROCEDURE InitEventLoop ( el : EventLoop );
-- Initialises the EventLoop object.
```

1.18 Using the EventLoop class

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 Collect() method must be called to enable garbage collection and specify its frequency. The value passed in the 'collectFreq' parameter indicates the number of signals that must be received between activations of the garbage collector. The higher the value, the less frequent the activations. Garbage collection can be disabled by passing 'NoGC' as the parameter. It is disabled by default when an EventLoop object is initialised by InitEventLoop().

The Do() 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 by 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'. If garbage collection is activated, care must be taken to make sure that there are no 'live' local pointer variables when the Do() is called. See the section on Garbage~Collection in OC.doc.

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

1.19 EventLoop class example

```
...
VAR
  sig, ignore : Signal;
  mp : MessagePort;
  ip : IdcmpPort;
  eventLoop : EventLoop;
...
BEGIN
  ...
  NEW (eventLoop); InitEventLoop (eventLoop);
  ...
```

```
  NEW (sig); NEW (mp); NEW (ip);
  ...
  ignore := eventLoop.AddSignal (sig);
  ignore := eventLoop.AddSignal (mp);
  ignore := eventLoop.AddSignal (ip);
  ...
  eventLoop.Collect (100); (* Activate garbage collection *)
  eventLoop.Do;
  ...
  (* Clean up *)
  ...
END
```