ISupEvents.mod

COLLABORATORS			
	TITLE : ISupEvents.mod		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		August 30, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

ISupEvents.mod

1.1 ISupEvents

```
MODULE ISupEvents;
(*
     $RCSfile: ISupEvents.mod $
  Description: Implementation of IntuiSup classes

   Created by: fjc (Frank Copeland)
    $Revision: 1.5 $
      $Author: fjc $
        $Date: 1994/09/03 16:25:44 $

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

  Module ISupEvents extends the classes defined in Module Events to
  provide the event management required by the IntuiSup library, written
  as an alternative to GadTools library by Torsten Jürgeleit.

   Overview   Overview of IntuiSup event handling

  Class descriptions

   ISupPort     Description of the ISupPort Class
   ISupDialog   Description of the ISupDialog Class

  Other

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

1.2 Overview of IntuiSup event handling

IntuiSup was created as an alternative to Commodore's GadTools library. It provides almost identical facilities (plus a few extra ones, such as a form of locale support), but unlike GadTools it will work under Kickstart 1.3.

Like GadTools, IntuiSup requires programmers to replace calls to Exec's GetMsg() and ReplyMsg() functions with calls to its own functions (see GTEvents). Unlike GadTools, IntuiSup in some cases sends the Task a special form of IntuiMessage which must be interpreted differently to an ordinary IntuiMessage. Special IntuiSup messages are identified by the string "ISUP" in the Class field and the other fields of the message have their meanings altered.

The ISupPort class is an extension of the IdcmpPort class which overrides the methods that rely on the Exec functions and replaces them with methods that use the IntuiSup functions. It also adds a new method to deal with the specially formatted IntuiSup messages.

IntuiSup also provides a RequesterData structure which can be used to create requesters that make full use of the facilities of IntuiSup library. The ISupDialog class is an extension of the RequesterData type that associates it with an ISupPort object. The ISupPort object is responsible for user interaction with the requester.

1.3 Revision History

```
$Revision: 1.5 $
    $Date: 1994/09/03 16:25:44 $
```

```
$Log: ISupEvents.mod $
Revision 1.5  1994/09/03  16:25:44  fjc
*** empty log message ***

Revision 1.4  1994/08/08  16:20:15  fjc
Release 1.4

Revision 1.3  1994/06/14  02:06:07  fjc
- Updated for release

Revision 1.2  1994/06/04  15:50:17  fjc
- Changed to use new Amiga interface

Revision 1.1  1994/05/12  19:57:07  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 turned off and replaced with ASSERTs at the appropriate
  places.
```

1.5 Imported Modules

```
IMPORT
  E := Exec, I := Intuition, ISup := IntuiSup, Ev := Events,
  SYS := SYSTEM;

(*
   Exec        Exec library
   Intuition   Intuition library
   IntuiSup    IntuiSup library
   Events      Module Events
```

1.6 Description of ISupPort

```
  A ISupPort is basically an IdcmpPort that has been modified to use the
  IntuiSup IGetMsg() and IReplyMsg() functions.  An additional method,
  HandleISup(), has been added to deal with specially formatted IntuiSup
  messages.
  ____________________________________________________________________________

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

1.7 Using ISupPort

```
  The ISupPort class is an abstract class that must be extended to
  create a concrete class before it can be used.  The extended class
  must at a minimum override the HandleISup() method.

  An ISupPort object may be used anywhere an IdcmpPort object would be
  appropriate.  It is initialised in the same way as an IdcmpPort
  object, mainly by installing handler procedures in the object's Handle
  field.

  The only other requirement is to use IntuiSup.IReplyMsg() instead of
  Exec.ReplyMsg() in handler procedures that consume messages.
  ____________________________________________________________________________

   Example   Example of using ISupPort
```

1.8 Example of using ISupPort

THIS SPACE INTENTIONALLY LEFT BLANK

1.9 Description of ISupDialog

An ISupDialog object is used to associate an ISupPort object with an IntuiSup.RequesterData structure. This allows the programmer to combine IntuiSup library's automatic requester management with the simplified event handling offered by Modules Events and ISupEvents. A single method, Activate(), takes care of all the details of displaying a modal requester and cleaning up afterwards.

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

1.10 Using ISupDialog

The ISupDialog class is best treated as an abstract class and extended to create a concrete class before it is used. A quick and dirty program could treat it as a concrete class if necessary.

An extension of ISupDialog would usually add a number of IntuiSup GadgetData, TextData and/or BorderData structures that provide the imagery of the requester. An initialisation method for the extended class would set up these structures, as well as assigning an ISupPort object to the iSupPort field.

It will normally be necessary to create an extension of the ISupPort class to do the event handling for the ISupDialog. This must override the HandleISup method to process any interaction with the requester's gadgets. If the ISupPort methods need to access the fields of the extended ISupDialog, the programmer must declare a field of the extended ISupDialog's type in the extended ISupPort and assign the ISupDialog object to it during initialisation.

If it is necessary to pass data to and from the ISupDialog (and it normally will be), the programmer must override the Activate method and write a replacement that will carry out the necessary processing.

Example	Example of using ISupDialog

1.11 ISupPort Declarations

```
TYPE
```

```
  ISupPort *= POINTER TO ISupPortRec;
  ISupPortRec *= RECORD (Ev.IdcmpPortRec) END;

(*
```

1.12 ISupDialog Declarations

```
(*
  ISupDialog object fields:

  iSupPort -- the ISupPort object used to control the requester.
*)

TYPE

  ISupDialog *= POINTER TO ISupDialogRec;
  ISupDialogRec *= RECORD (ISup.RequesterData)
    iSupPort *: ISupPort;
  END;

(*
```

1.13 ISupPort Methods

```
  OVERRIDDEN METHODS

  Event handling

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

    PROCEDURE ( isp : ISupPort ) HandleMsg* ( msg ) : INTEGER;
    - Checks for specially formatted IntuiSup messages and determines
      which message handler is to be used.

  Message Port maintenance

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

  NEW METHODS

  Event handling

    PROCEDURE ( isp : ISupPort ) HandleISup* ( msg ) : INTEGER;
    - Handles specially formatted IntuiSup messages.
```

1.14 HandleSig()

```
PROCEDURE (isp : ISupPort) 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 isp.HandleMsg().
*)

  VAR result : INTEGER; msg : I.IntuiMessagePtr;

BEGIN (* HandleSig *)
  result := Ev.Pass; (* Default *)
  (* Loop until all messages queued at the port are dealt with *)
  ASSERT (isp.port # NIL, 132);
  LOOP
    (* Dequeue the next message, quit if there is none *)
    msg := ISup.base.GetMsg (isp.port);
    IF msg = NIL THEN EXIT END;
    (* Despatch to the message handler *)
    result := isp.HandleMsg (msg);
    (* Process the return code *)
    IF result = Ev.Pass THEN ISup.base.ReplyMsg (msg) END;
    IF result > Ev.Continue THEN EXIT END
  END;
  RETURN result
END HandleSig;
(*
```

1.15 HandleISup()

```
PROCEDURE (isp : ISupPort) HandleISup* (msg : I.IntuiMessagePtr) : INTEGER;
(*
  Each descendant class is expected to override this method and provide
  its own implementation.  In order for the Events.SimpleLoop (and
  EventLoop.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 'Events.Pass'.
  - If the ISupPort no longer needs to be part of the event loop (ie-
    when a Window is closed), it must return 'Events.Stop'.
  - If the event causes the program to terminate, the method must
    return 'Events.StopAll'.
  - In all other cases it must return 'Events.Continue'.

  If the method returns any result other than 'Pass' it must first call
  'ISup.base.ReplyMsg (msg)' to remove the Message from the MsgPort.
  If it returns 'Pass', it should *never* call IReplyMsg().
*)

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

1.16 HandleMsg()

```
PROCEDURE (isp : ISupPort) HandleMsg* ( msg : E.MessagePtr ) : INTEGER;
(*
  This method overrides the method in IdcmpPort.  It checks to see if
  'msg' is a special IntuiSup message.  If it is, it despatches to the
  HandleISup() method to deal with it.  Otherwise, it despatches to the
  supermethod defined by IdcmpPort.
*)

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

BEGIN (* HandleMsg *)
  (* Type cast message to an IntuiMessagePtr *)
  intuiMessage := SYS.VAL (I.IntuiMessagePtr, msg);
  (* Work out who will do the work *)
  IF intuiMessage.class = SYS.VAL (SET, ISup.id) THEN
    RETURN isp.HandleISup (intuiMessage)
  ELSE
    RETURN isp.HandleMsg^ (msg)
  END
END HandleMsg;
(*
```

1.17 FlushPort()

```
PROCEDURE (isp : ISupPort) 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 : I.IntuiMessagePtr;

BEGIN (* FlushPort *)
  ASSERT (isp.port # NIL, 132);
  E.base.Forbid ();
    LOOP
      msg := ISup.base.GetMsg (isp.port);
      IF msg = NIL THEN EXIT END;
      ISup.base.ReplyMsg (msg)
    END;
  E.base.Permit ()
END FlushPort;
(*
```

1.18 ISupDialog Methods

```
  NEW METHODS

  Activation

    PROCEDURE Activate* ( isd, window ) : BOOLEAN;
```

1.19 Activate()

```
PROCEDURE Activate* (isd : ISupDialog; window : I.WindowPtr) : BOOLEAN;
(*
  This method displays an IntuiSup requester in a window as a modal
  requester. This means that the window's message port becomes inactive
  and no messages will be sent to it as long as the requester is
  displayed.  It returns TRUE if the requester is successfully displayed,
  otherwise it returns FALSE.  It saves the window's IDCMP flags and
  restores them before exiting (IntuiSup appears not to do this itself).
*)

  VAR
    savedIDCMP : SET; rList : ISup.RequesterListPtr; iSupPort : ISupPort;
    reqWin : I.WindowPtr;

BEGIN (* Activate *)
  ASSERT (isd # NIL, 132);
  rList := ISup.base.DisplayRequester (window, isd^, NIL);
  IF rList # NIL THEN
    savedIDCMP := window.idcmpFlags;
    iSupPort := isd.iSupPort;
    reqWin := rList.reqWindow;
    Ev.AttachPort (iSupPort, reqWin.userPort);
    iSupPort.SetupWindow (reqWin);
    Ev.SimpleLoop (iSupPort);
    iSupPort.CleanupWindow (reqWin);
    Ev.DetachPort (iSupPort);
    ISup.base.RemoveRequester (rList);
    I.base.OldModifyIDCMP (window, savedIDCMP);
    RETURN TRUE
  END;
  RETURN FALSE
END Activate;
(*
```

1.20 Module Initialisation

```
(* No initialisation required *)

END ISupEvents.
(*
```