Libraries

COLLABORATORS

	TITLE : Libraries		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		March 14, 2022	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Libraries

1.1 Amiga® RKM Libraries: 24 Exec Messages and Ports

For interprocess communication, Exec provides a consistent, high-performance mechanism of messages and ports. This mechanism is used to pass message structures of arbitrary sizes from task to task, interrupt to task, or task to software interrupt. In addition, messages are often used to coordinate operations between cooperating tasks. This chapter describes many of the details of using messages and ports that the casual Amiga programmer won't need. See the "Introduction to Exec" chapter of this manual for a general introduction to using messages and ports.

A message data structure has two parts: system linkage and message body. The system linkage is used by Exec to attach a given message to its destination. The message body contains the actual data of interest. The message body is any arbitrary data up to 64K bytes in size. The message body data can include pointers to other data blocks of any size.

Messages are always sent to a predetermined destination port. At a port, incoming messages are queued in a first-in-first-out (FIFO) order. There are no system restrictions on the number of ports or the number of messages that may be queued to a port (other than the amount of available system memory).

Messages are always queued by reference, i.e., by a pointer to the message. For performance reasons message copying is not performed. In essence, a message between two tasks is a temporary license for the receiving task to use a portion of the memory space of the sending task; that portion being the message itself. This means that if task A sends a message to task B, the message is still part of the task A context. Task A, however, should not access the message until it has been replied; that is, until task B has sent the message back, using the
ReplyMsg()
function.
This technique of message exchange imposes important restrictions on message access.

Message Ports

Messages

Function Reference

1.2 24 Exec Messages and Ports / Message Ports

Message ports are rendezvous points at which messages are ↩ collected. A
port may contain any number of outstanding messages from many different
originators. When a message arrives at a port, the message is appended to
the end of the list of messages for that port, and a prespecified arrival
action is invoked. This action may do nothing, or it may cause a
predefined task signal or software interrupt (see the "Exec Interrupts"
chapter).

Like many Exec structures, ports may be given a symbolic name. Such names
are particularly useful for tasks that must rendezvous with dynamically
created ports. They are also useful for debugging purposes.

A message port consists of a MsgPort structure as defined in the
<exec/ports.h> and <exec/ports.i> include files. The C structure for a
port is as follows:

```
    struct MsgPort {
        struct Node  mp_Node;
        UBYTE        mp_Flags;
        UBYTE        mp_SigBit;
        struct Task *mp_SigTask;
        struct List  mp_MsgList;
    };
```

mp_Node
    is a standard Node structure. This is useful for tasks that might
    want to rendezvous with a particular message port by name.

mp_Flags
    are used to indicate message arrival actions. See the explanation
    below.

mp_SigBit
    is the signal bit number when a port is used with the task signal
    arrival action.

mp_SigTask
    is a pointer to the task to be signaled. If a software interrupt
    arrival action is specified, this is a pointer to the interrupt
    structure.

mp_MsgList
    is the list header for all messages queued to this port. (See the
    "Exec Lists and Queues" chapter).

The mp_Flags field contains a subfield indicated by the PF_ACTION mask.
This sub-field specifies the message arrival action that occurs when a
port receives a new message. The possibilities are as follows:

PA_SIGNAL
    This flag tells Exec to signal the mp_SigTask using signal number
    mp_SigBit on the arrival of a new message. Every time a message is
    put to the port another signal will occur regardless of how many
    messages have been queued to the port.

PA_SOFTINT
    This flag tells Exec to Cause() a software interrupt when a message
    arrives at the port. In this case, the mp_SigTask field must contain
    a pointer to a struct Interrupt rather than a Task pointer. The
    software interrupt will be Caused every time a message is received.

PA_IGNORE
    This flag tells Exec to perform no operation other than queuing the
    message. This action is often used to stop signaling or software
    interrupts without disturbing the contents of the mp_SigTask field.

It is important to realize that a port's arrival action will occur for each new message queued, and that there is not a one-to-one correspondence between messages and signals. Task signals are only single-bit flags so there is no record of how many times a particular signal occurred. There may be many messages queued and only a single task signal; sometimes however there may be a signal, but no messages. All of this has certain implications when designing code that deals with these actions. Your code should not depend on receiving a signal for every message at your port. All of this is also true for software interrupts.

1.3 24 / Message Ports / Creating a Message Port

To create a new message port using an operating system release ↩
prior to
V36, you must allocate and initialize a
MsgPort
structure. If you want to
make the port public, you will also need to call the AddPort() function.
Don't make a port public when it is not necessary for it to be so. The easiest way to create a port is to use the amiga.lib function CreatePort(name,priority). If NULL is passed for the name, the port will not be made public. Port structure initialization involves setting up a Node structure, establishing the message arrival action with its parameters, and initializing the list header. The following example of port creation is equivalent to the CreatePort() function as supplied in amiga.lib:

```
    struct MsgPort *CreatePort(UBYTE *name, LONG pri)
    {
        LONG sigBit;
        struct MsgPort *mp;

        if ((sigBit = AllocSignal(-1L)) == -1) return(NULL);

        mp = (struct MsgPort *) AllocMem((ULONG)sizeof(struct MsgPort),
                 (ULONG)MEMF_PUBLIC | MEMF_CLEAR);
        if (!mp) {
            FreeSignal(sigBit);
            return(NULL);
        }
        mp->mp_Node.ln_Name = name;
        mp->mp_Node.ln_Pri  = pri;
        mp->mp_Node.ln_Type = NT_MSGPORT;
        mp->mp_Flags        = PA_SIGNAL;
        mp->mp_SigBit       = sigBit;
        mp->mp_SigTask      = (struct Task *)FindTask(0L);
                                                  /* Find THIS task.   */

        if (name) AddPort(mp);
        else NewList(&(mp->mp_MsgList));          /* init message list */

        return(mp);
    }
```

As of V36 the Exec CreateMsgPort() function can be used to create a message port. This function allocates and initializes a new message port. Just like CreatePort(), a signal bit will be allocated and the port will be initialized to signal the creating task (
mp_SigTask
) when a message
arrives at this port. To make the port public after CreateMsgPort(), you must fill out the ln_Name field and call AddPort(). If you do this, you must remember to
RemPort()
 the port from the public list in your cleanup.
If you need to create a message port and your application already requires Release 2 or greater, you can use CreateMsgPort() instead of CreatePort(). The following is an example of the usage of CreateMsgPort().

```
    struct MsgPort *newmp;
            /* A private message port has been created. CreateMsgPort() */
    if (newmp = CreateMsgPort())
            /* returns NULL if the creation of the message port failed. */
    {
        newmp->mp_Node.ln_Name = "Griffin";
        newmp->mp_Node.ln_Pri  = 0;
                                 /* To make it public fill in the fields */
        AddPort(newmp);          /* with appropriate values.             */
    }
```

1.4 24 / Message Ports / Deleting a Message Port

Before a message port is deleted, all outstanding messages from ↩
other
tasks must be returned. This is done by getting and replying to all
messages at the port until message queue is empty. Of course, there is no
need to reply to messages owned by the current task (the task performing
the port deletion). Public ports attached to the system with
AddPort()
must be removed from the system with RemPort() before deallocation ↩
. This
amiga.lib functions
CreatePort()
and DeletePort() handle this
automatically.

The following example of port deletion is equivalent to the DeletePort()
function as supplied in amiga.lib. Note that DeletePort() must only be
used on ports created with
CreatePort()
.

```
    void DeletePort(mp)
    struct MsgPort *mp;
    {
        if ( mp->mp_Node.ln_Name ) RemPort(mp);  /* if it was public... */

        mp->mp_SigTask         = (struct Task *) -1;
                                 /* Make it difficult to re-use the port */
        mp->mp_MsgList.lh_Head = (struct Node *) -1;

        FreeSignal( mp->mp_SigBit );
        FreeMem( mp, (ULONG)sizeof(struct MsgPort) );
    }
```

To delete ports created with
CreateMsgPort()
, DeleteMsgPort() must be
used. Note that these functions are only available in V36 and higher. If
the port was made public with
AddPort()
, RemPort() must be used first, to
remove the port from the system. Again, make sure all outstanding
messages are replied to, so that the message queue is empty.

```
    struct MsgPort *newmp;

    if (newmp)
    {
        if ( newmp->mp_Node.ln_Name ) RemPort(newmp);
                                 /* if it was public... */
        DeleteMsgPort(newmp);
    }
```

1.5 24 / Message Ports / How To Rendezvous at a Message Port

The FindPort() function provides a means of finding the address of a public port given its symbolic name. For example, FindPort("Griffin") will return either the address of the message port named "Griffin" or NULL indicating that no such public port exists. Since FindPort() does not do any arbitration over access to public ports, the usage of FindPort() must be protected with Forbid()/Permit(). Names should be unique to prevent collisions among multiple applications. It is a good idea to use your application name as a prefix for your port name. FindPort() does not arbitrate for access to the port list. The owner of a port might remove it at any time. For these reasons a Forbid()/Permit() pair is required for the use of FindPort(). The port address can no longer be regarded as being valid after Permit() unless your application knows that the port cannot go away (for example, if your application created the port).

The following is an example of how to safely put a message to a specific port:

```
    #include <exec/types.h>
    #include <exec/ports.h>

       BOOL MsgPort SafePutToPort(struct Message *message, STRPTR portname)
       {
           struct MsgPort *port;

           Forbid();
           port = FindPort(portname);
           if (port) PutMsg(port,message);
           Permit();
           return(port ? TRUE : FALSE);
                            /* If FALSE, the port was not found */

           /* Once we've done a Permit(), the port might go away */
           /* and leave us with an invalid port address. So we   */
           /* return just a BOOL to indicate whether the message */
           /* has been sent or not.                              */
       }
```

1.6 24 Exec Messages and Ports / Messages

As mentioned earlier, a message contains both system header ↩
information
and the actual message content. The system header is of the Message form defined in <exec/ports.h> and <exec/ports.i>. In C this structure is as follows:

```
    struct Message {
        struct Node     mn_Node;
        struct MsgPort *mn_ReplyPort;
        UWORD           mn_Length;
    };
```

mn_Node
    is a standard Node structure used for port linkage.

```
mn_ReplyPort
    is used to indicate a port to which this message will be returned
    when a reply is necessary.

mn_Length
    indicates the total length of the message, including the Message
    structure itself.
```

This structure is always attached to the head of all messages. For example, if you want a message structure that contains the x and y coordinates of a point on the screen, you could define it as follows:

```
    struct XYMessage {
        struct Message xy_Msg;
        UWORD          xy_X;
        UWORD          xy_Y;
    }
```

For this structure, the mn_Length field should be set to sizeof(struct XYMessage).

1.7 24 / Messages / Putting a Message

A message is delivered to a given destination port with the PutMsg ↩
() function. The message is queued to the port, and that port's arrival action is invoked. If the action specifies a task signal or a software interrupt, the originating task may temporarily lose the processor while the destination processes the message. If a reply to the message is required, the
mn_ReplyPort
field must be set up prior to the call to
PutMsg().

Here is a code fragment for putting a message to a public port. A complete example is printed at the end of the chapter.

```
#Include <exec/types.h>
#include <exec/memory.h>
#include <exec/ports.h>
#include <libraries/dos.h>

VOID main(VOID);
BOOL SafePutToPort(struct Message *, STRPTR);
```

```
struct XYMessage {
    struct Message xy_Msg;
    UWORD          xy_X;
    UWORD          xy_Y;
};

VOID main(VOID)
{
    struct MsgPort *xyport, *xyreplyport;
    struct XYMessage *xymsg, *msg;
    BOOL   foundport;

    /* Allocate memory for the message we're going to send. */
    if (xymsg = (struct XYMessage *) AllocMem(sizeof(struct XYMessage),
                                         MEMF_PUBLIC | MEMF_CLEAR))
    {

        /* The replyport we'll use to get response */
        if (xyreplyport = CreateMsgPort()) {
                                       /* or use CreatePort(0,0) */

            xymsg->xy_Msg.mn_Node.ln_Type = NT_MESSAGE;
                                       /* Compose the message    */
            xymsg->xy_Msg.mn_Length = sizeof(struct XYMessage);
            xymsg->xy_Msg.mn_ReplyPort = xyreplyport;
            xymsg->xy_X = 10;
            xymsg->xy_Y = 20;

            /* Now try to send that message to a public port named
             *  "xyport". If foundport eq 0, the port isn't out there.
             */
            if (foundport = SafePutToPort((struct Message *)xymsg,
                                          "xyport"))
            {

            . . .   /* Now let's wait till the someone responds... */

            }
            else printf("Couldn't find 'xyport'\n");

            DeleteMsgPort(xyreplyport);  /* Use DeletePort() if    */
                                       /* the port was created   */
        }                              /* with CreatePort().     */
        else printf("Couldn't create message port\n");
        FreeMem(xymsg, sizeof(struct XYMessage));
    }
    else printf("Couldn't get memory for xymessage\n");
}
```

1.8 24 / Messages / Waiting For a Message

```
                A task may go to sleep waiting for a message to arrive at one or ←
                   more
```

ports. This technique is widely used on the Amiga as a general form of
event notification. For example, it is used extensively by tasks for I/O
request completion.

The
MsgPort.mp_SigTask
field contains the address of the task to be
signaled and
mp_SigBit
contains a preallocated signal number (as described
in the "Exec Tasks" chapter).

You can call the WaitPort() function to wait for a message to arrive at a
port. This function will return the first message (it may not be the
only) queued to a port. Note that your application must still call

GetMsg()
to remove the message from the port. If the port is empty, your
task will go to sleep waiting for the first message. If the port is not
empty, your task will not go to sleep. It is possible to receive a signal
for a port without a message being present yet. The code processing the
messages should be able to handle this. The following code illustrates
WaitPort().

```
    struct XYMessage *xy_msg;
    struct MsgPort   *xyport;

    xyport = CreatePort("xyport", 0);
    if (xyport == 0)
    {
        printf("Couldn't create xyport\n");
        exit(31);
    }

    xy_msg = WaitPort(xyport);     /* go to sleep until message arrives */
```

A more general form of waiting for a message involves the use of the
Wait() function (see the "Exec Signals" chapter). This function waits for
task event signals directly. If the signal assigned to the message port
occurs, the task will awaken. Using the Wait() function is more general
because you can wait for more than one signal. By combining the signal
bits from each port into one mask for the Wait() function, a loop can be
set up to process all messages at all ports.

Here's an example using Wait():

```
    struct XYMessage *xy_msg;
    struct MsgPort   *xyport;
    ULONG usersig, portsig;
    BOOL ABORT = FALSE;

    if (xyport = CreatePort("xyport", 0))
    {
        portsig = 1 << xyport->mp_SigBit;
        usersig = SIGBREAKF_CTRL_C;     /* User can break with CTRL-C.  */
        for (;;)
        {
            signal = Wait(portsig | usersig);
                                        /* Sleep till someone signals.  */

            if (signal & portsig)       /* Got a signal at the msgport. */
            {   .  .  .
            }
            if (signal & usersig)       /* Got a signal from the user.  */
            {
                ABORT = TRUE;           /* Time to clean up.            */
                 . . .
            }
            if (ABORT) break;
        }
        DeletePort(xyport);
    }
    else printf("Couldn't create xyport\n");


    WaitPort() Does Not Remove A Message.
    -------------------------------------
    WaitPort() only returns a pointer to the first message in a port.
    It does not actually remove the message from the port queue.
```

1.9 24 / Messages / Getting a Message

Messages are usually removed from ports with the GetMsg() function. This function removes the next message at the head of the port queue and returns a pointer to it. If there are no messages in a port, this function returns a zero.

The example below illustrates the use of GetMsg() to print the contents of all messages in a port:

```
    while (xymsg = GetMsg(xyport)) printf("x=%ld y=%ld\n", xymsg->xy_X,
                                            xymsg->xy_Y);
```

Certain messages may be more important than others. Because ports impose FIFO ordering, these important messages may get queued behind other messages regardless of their priority. If it is necessary to recognize more important messages, it is easiest to create another port for these special messages.

1.10 24 / Messages / Replying

When the operations associated with receiving a new message are ↩ finished,
it is usually necessary to send the message back to the originator. The receiver replies the message by returning it to the originator using the ReplyMsg() function. This is important because it notifies the originator that the message can be reused or deallocated. The ReplyMsg() function serves this purpose. It returns the message to the port specified in the

mn_ReplyPort
field of the message. If this field is zero, no reply is returned.

The
previous example
can be enhanced to reply to each of its messages:

```
    while (xymsg = GetMsg(xyport)) {
        printf("x=%ld y=%ld\n", xymsg->xy_X, xymsg->xy_Y);
        ReplyMsg(xymsg);
    }
```

Notice that the reply does not occur until after the message values have been used.

Often the operations associated with receiving a message involve returning results to the originator. Typically this is done within the message itself. The receiver places the results in fields defined (or perhaps reused) within the message body before replying the message back to the originator. Receipt of the replied message at the originator's reply port indicates it is once again safe for the originator to use or change the values found within the message.

The following are two short example tasks that communicate by sending, waiting for and replying to messages. Run these two programs together.

Port1.c   Port2.c

1.11 24 Exec Messages and Ports / Function Reference

The following chart gives a brief description of the Exec functions that control inter-task communication with messages and ports. See the Amiga ROM Kernel Reference Manual: Includes and Autodocs for details about each call.

Table 24-1: Exec Message and Port Functions

Function	Description
AddPort()	Add a public message port to the system list.
CreateMsgPort()	Allocate and initialize a new message port (V37).
DeleteMsgPort()	Free a message port, created with CreateMsgPort() (V37).
FindPort()	Find a public message port in the system list.
GetMsg()	Get next message from the message port.
PutMsg()	Put a message to a message port.
RemPort()	Remove a message port from the system list.
ReplyMsg()	Reply to a message on its reply port.

Table 24-2: Amiga.lib Exec Support Functions

Function	Description
CreatePort()	Allocate and initialize a new message port, make public if named
DeletePort()	Delete a message port, created with CreatePort().