AmigaMail

COLLABORATORS

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

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

AmigaMail

1.1   II-95: Packet Level I/O under Release 2

Packet Level I/O under Release 2

by Dale Larson and John Orr

Normally when an application needs to perform I/O using DOS, it uses
functions from the dos.library to open, read, write, and close any
I/O channels. These functions take care of talking to the underlying
DOS device (see the ``About Devices'' section below), shielding the
programmer from most of the grunt work it takes to carry out the I/O.
These functions are adequate for most I/O needs.

```
 **********************************************************
 *                                                        *
 * About Devices                                          *
 *                                                        *
 *                                                        *
 * The term ``DOS devices'' is confusing because the      *
 * term ``device'' is a bit overloaded.  There are        *
 * three entities on the Amiga that are referred to as    *
 * a device.  There is the Exec device (for example       *
 * trackdisk.device and console.device) the AmigaDOS      *
 * device (such as DF0: and RAM:), and the logical        *
 * AmigaDOS device name (for example, SYS:, C:, and       *
 * S:).  The Exec level device is basically a subset of   *
 * an Exec library.  Exec devices are described in the    *
 * RKRM: Devices book. The AmigaDOS device is a higher    *
 * level entity than the Exec device (AmigaDOS devices    *
 * often utilize Exec devices).  An AmigaDOS device has   *
 * a process associated with it, normally referred to     *
 * as a handler, that controls the AmigaDOS I/O stream    *
 * via a FileHandle.  The AmigaDOS logical device name    *
 * (better known as an ``assign'') is made to look like   *
 * an AmigaDOS device, but is a higher level entity       *
 * than the AmigaDOS device.  Both the AmigaDOS device    *
 * and the logical device name are referred to by a       *
 * name ending in a colon (':').  The logical device      *
```

```
     * can refer to any directory on an AmigaDOS device (as   *
     * long as the AmigaDOS device supports having files).    *
     * The user can change the directory to which logical     *
     * device name refers from a command line. This article   *
     * deals primarily with AmigaDOS devices.                 *
     *                                                         *
     ***********************************************************
```

One of the reasons these functions are only adequate is because they
are synchronous. When an application attempts to transfer data using
one of these functions, the I/O function waits for the entire
transfer to finish before returning. Ideally, the application should
be able to perform I/O asynchronously, so it can do something else
while the data transfer takes place.

Another reason that these functions are only adequate is because they
don't implement all of the features built into DOS devices. To
utilize these features, an application has to work on a lower level.
The application has to talk directly to the DOS device.

The Packet Level

When DOS functions talk to devices such as the floppy drive or the
serial port, they talk to a special process called a handler. Every
DOS device (like CON:, SER:, DF0:, and PIPE:) has a handler process.
The handler process is responsible for receiving and carrying out a
standard set of DOS device commands. It provides a standard
programming interface to a lower-level I/O software or hardware
entity (such as an Exec device). The packet interface abstracts the
lower-level entity so that dos.library functions don't have to deal
with a lot of bothersome details such as moving a disk head or
reading serial registers.   The interface to every handler is the
same, so every DOS device operates in the same manner, regardless of
any underlying software or hardware. Theoretically, to the
dos.library functions, writing to the console handler (CON:) should
be no different than writing to the serial port handler (SER:) or the
DF0: handler. The handler lets dos.library functions treat all DOS
devices in the same way.

Typically, the handler talks directly to an underlying Exec device.
Two examples are the CON: and the DF0: handlers. The CON: handler
process talks directly to the console.device.   When trying to access
a floppy in DF0:, the DF0: handler talks directly to the
trackdisk.device. These handlers accept handler level commands (for
example, ACTION_READ and ACTION_WRITE) and hide any Exec level I/O
from the dos.library functions and subsequently, the application.

In cases like the DF0: handler, which is a special type of handler
called a file system, the handler has to take care of organizing the
lower-level medium into directories and files. The DF0: handler
takes care of translating directory and file names into terms the
trackdisk.device can understand (disk blocks). The only requirement
of a handler to be a file system is that the handler must support
files. A file system does not have to support a directory structure
to be considered a file system. Handlers which are not file systems
are called stream handlers.

Some handlers do not have any underlying software or hardware.
Handlers such as RAM: have no underlying software or hardware. These
kinds of handlers take care of implementing everything necessary to
carry out the I/O rather than delegating to an Exec device.

Handlers receive commands through an Exec message port. Every
handler process has a message port for receiving commands. A handler
accepts a command in the form of a DosPacket structure (defined in
<dos/dosextens.h>), which is an extension of an Exec Message
structure:

```
struct DosPacket {
    struct Message *dp_Link;
    struct MsgPort *dp_Port;  /* Reply port for the packet */
                              /* Must be filled in each send. */
    LONG dp_Type;
    LONG dp_Res1;             /* Result #1 */
    LONG dp_Res2;             /* For file system calls this is what would
                               * have been returned by IoErr() */
    LONG dp_Arg1;             /* Argument list */
    LONG dp_Arg2;
    LONG dp_Arg3;
    LONG dp_Arg4;
    LONG dp_Arg5;
    LONG dp_Arg6;
    LONG dp_Arg7;
}; /* DosPacket */
```

The dp_Type field contains an identifier corresponding to the
command. The identifiers for each of the standard commands are
defined in <dos/dosextens.h>. For example, the command to write data
is ACTION_WRITE. Each packet type has different parameters, which
the programmer supplies in the ``dp_Arg'' fields.

After a handler finishes with a packet, it returns the packet to the
message port in dp_Port. The handler places return values (including
any error codes) in the result fields dp_Res1 and dp_Res2. If there
was an error, dp_Res2 contains the corresponding DOS error code.

The packet types are described in the Amiga Mail article ``AmigaDOS
Packet Interface Specification'' on page II-5 and also in The
AmigaDOS Manual, 3rd Edition. Since its original publication, the
``AmigaDOS Packet Interface Specification'' has been updated numerous
times in Amiga Mail to correct errors. The information in that
article (plus its errata) should appear in the next edition of The
AmigaDOS Manual.

Finding the Handler

There are various ways to find the address of the target handler's
Message port, which is also called a process identifier by some
documentation. When working with an open FileHandle, the handler's
port address is in the FileHandle's fh_Type field. Note that the
dos.library functions normally access a FileHandle using a BPTR, so
to get to the fh_Type field an application has to do something like

this:

```
    my_handler_port = ((struct FileHandle *)BADDR(FileHandle))->fh_Type);
```

Because the AmigaDOS device NIL: has no handler process, the fh_Type
field of any of its file handle's will be NULL. The application must
account for this case.

When working with a device or assign name, an application can get to
the handler's message port by using the dos.library function
GetDeviceProc():

```
    struct DevProc *GetDeviceProc(UBYTE *dev_name, struct DevProc *prev_devproc);
```

This function returns a pointer to the following structure:

```
struct DevProc {
        struct MsgPort *dvp_Port;     /* The handler's Message port    */
        BPTR            dvp_Lock;     /*             (send packets there) */
        ULONG           dvp_Flags;
        struct DosList *dvp_DevNode;  /* DON'T TOUCH OR USE! */
};
```

This function is intended to improve on the DeviceProc() function as
it also deals with multiple assigns. GetDeviceProc() must be
countered by FreeDeviceProc(). See the GetDeviceProc() Autodoc for
more details.

The Old Way

Prior to Release 2, sending packets was relatively complicated. Some
of the early disks from the Fred Fish Library (disks 35, 56, and 66)
contain complete examples of using DOS packets synchronously and
asynchronously. As of Release 2.04, dos.library contains functions
which make it easier to use DOS packets synchronously and
asynchronously.

Synchronous Packet Calls

Performing synchronous packet I/O is now very simple. Simply call
the dos.library function DoPkt():

```
LONG DoPkt(struct MsgPort *handler_port, LONG action_id, LONG arg1,
          LONG arg2, LONG arg3, LONG arg4, LONG arg5);
```

This function allocates and fills out a DosPacket using the packet
type ('action_id') and arguments ('arg1', 'arg2', etc.) you supply.
This function then sends off the packet to the 'handler_port' and
waits for the packet to return.   DoPkt() returns the value from the
packet's dp_Res1 field. To get the value from the packet's dp_Res2
field, call the dos.library function IoErr(). Assembly language
programmers can also get dp_Res2 from register D1.

Asynchronous Packet Calls

To do asynchronous packet calls, things are a little more complex,
but they are still much better than they were before V37 (This
subject was partially covered in Martin Taillefer's article, ''Fast
AmigaDOS I/O'', page II-77, from the September/October 1992 issue of
Amiga Mail). When using a DOS packet asynchronously, there are three
things to do before sending the packet anywhere:

    o Acquire a message port
    o Allocate and initialize the packet
    o Set up the packet's arguments

Acquiring the Message Port

It is possible for an application to use its process message port for
packet transmissions rather than allocating a new one. It can get to
its Process structure by calling FindTask() with an argument of NULL.
The Process structure has an Exec MsgPort structure embedded in it,
which an application can get to via the pr_MsgPort field. The bad
thing about using this port is that many system functions also use
it. Consequently, after sending an asynchronous packet, an
application can not call any system functions that use pr_MsgPort.
This includes most dos.library functions, many C compiler linker
library functions, and many standard I/O functions (i.e., printf()
and scanf()). The application has to wait for the asynchronous
packet to return before calling such functions. If the application
sent an asynchronous packet and inadvertently initiated some other
packet level I/O before the asynchronous packet came back, it is
possible to cause an ''unexpected packet received'' system crash if
the asynchronous packet returned at the wrong time. Also, an
application should remove such a packet from the process message port
using the WaitPkt() function. This function will take care of any
system idiosyncrasies that the application would otherwise need to
address itself. This applies to idiosyncrasies that exist now or new
ones that may appear in the future.

As of V37, arguably the best and easiest way to acquire a message
port is to create one with the exec.library call CreateMsgPort(). By
allocating its own message port, the application doesn't have to
worry about any problems with sharing the port.

Allocating and Initializing a Packet

The packet I/O system uses Exec's message passing system, so each
DosPacket must also have an Exec Message structure. As both
structures are necessary, they have been combined in a single
StandardPacket structure (defined in <dos/dosextens.h>):

```
struct StandardPacket {
    struct Message   sp_Msg;
    struct DosPacket sp_Pkt;
}; /* StandardPacket */
```

To make a packet usable, the Exec Message and DosPacket structures
have to be set up to point to each other. The DosPacket structure is

straightforward about its link to its corresponding message. The
DosPacket's dp_Link field must point to the packet's Message
structure. However, the link from the Message to the DosPacket is a
bit obscure. The Message contains an Exec Node structure which in
turn contains a field called ln_Name. Although the Node structure
defines ln_Name as a character array, the DOS packet I/O system
instead requires that this field point to the Message's corresponding
DosPacket:

```
    my_standard_pkt->sp_Msg.mn_Node.ln_Name = (STRPTR)(my_standard_pkt->sp_Pkt);
```

As of V37, the dos.library function AllocDosObject() is usually the
best way to allocate a StandardPacket. To allocate one, call:

```
    mypacket = AllocDosObject(DOS_STDPKT, NULL);
```

This function takes care of linking the StandardPacket's Message and
DosPacket. Note that the function call above does not return a
pointer to a StandardPacket. This function allocates an entire
StandardPacket structure, but it returns a pointer to the
StandardPacket's sp_Pkt field. To access the sp_Msg portion of the
StandardPacket, use the pointer in the dp_Link field.

Any structure allocated with AllocDosObject() must be freed with
FreeDosObject(). To free 'mypacket' from the above AllocDosObject()
call:

```
    FreeDosObject(DOS_STDPKT, mypacket);
```

Filling in Packet Arguments

Before sending an asynchronous packet, an application needs to fill
in its action and arguments. For example, a packet set up to read
4096 bytes from an open file handle might look something like this:

. . .

```
#define BUFSIZE 4096
```

. . .

```
BPTR myfh;
struct DosPacket *my_pkt;
UBYTE *buffer[BUFSIZE];

        . . .

        my_pkt->dp_Type = ACTION_READ;
        my_pkt->dp_Arg1 = ((struct FileHandle *)BADDR(myfh))->fh_Arg1;
        my_pkt->dp_Arg2 = buffer;
        my_pkt->dp_Arg3 = BUFSIZE;

        . . .
```

Sending the Asynchronous Packet

To send a packet asynchronously, use the dos.library SendPkt() function:

```
SendPkt(struct DosPacket *mypacket,
        struct MsgPort *handlerport, struct MsgPort *replyport)
```

This function sends 'mypacket' to 'handlerport' and exits without
waiting for the packet to come back. When the packet returns, it
will arrive at 'replyport'.

Waiting for the Asynchronous Packet

After calling SendPkt(), the application can go about its business,
taking care of some other work while the DOS device handles the
application's packet. Eventually, the application will have to test
the reply port to see if the packet has come back yet. It can do
this with WaitPort() or, if the application has to test for more than
just the reply port's signal, it can use Wait(). If the application
doesn't poll its signals too often, it can test its own signal bits
using SetSignal() (see the exec.library Autodocs and the
``Introduction to Exec'' chapter of the Release 2 RKRM: Devices for
more information on how to use these functions). Be sure to remove
any and all packets from the message port using GetMsg().

If the application used its process message port (pr_MsgPort) as the
reply port, it must use the WaitPkt() function to remove the packet
from the message port. As mentioned earlier, this function will take
care of any hidden system idiosyncrasies so the application never has
to account for them. WaitPkt() will not necessarily clear the
process message port's signal bit. Like SendPkt(), WaitPkt() assumes
any DosPacket it works with is part of a StandardPacket structure.

The dos.library contains a function called AbortPkt() that, at a
glance, looks like it might be useful to an application that needs to
abort a packet (for example, upon receiving a break signal). This
function, which was introduced in Release 2, is supposed to attempt
to abort a packet that has already been sent to a handler. If the
abort operation is successful, the aborted packet will arrive at its
original reply port (the same place the packet would have arrived if
it was successful). Unfortunately, the abort operation will never be
successful, at least not under existing releases of the operating
system. Currently, this function returns without doing anything (the
most recent release is 3.0 or V39). In the future when AbortPkt()
does do something, it will assume that any DosPacket it works with is
part of a StandardPacket structure.

Interpreting the Packet Results

Upon returning to the application, for most packets, there will be
result values in dp_Res1 and dp_Res2. For most packets, if dp_Res1
is DOSTRUE, the packet returned without a problem. If dp_Res1 is
DOSFALSE, the handler experienced an error with the packet and there
should be an error code in dp_Res2. The error codes are defined in
<dos/dos.h>. Note that not all packets follow this error reporting
convention. See the ``AmigaDOS Packet Interface Specification''

article on page II-5 for more information on how individual packets
work.

Packets without dos.library Functions

Besides offering the ability to do asynchronous I/O, directly using
DOS packets also allows applications to utilize features of certain
handlers that are not available through a dos.library function.

The following packets are not available via a dos.library function as
of Release 3.0:

     ACTION_WRITE_PROTECT   ACTION_DISK_INFO (for console handlers)

If an application needs the feature that these packets provide, the
application has to send the packet to the handler. The first two
packets are fairly straightforward and are explained in the
``AmigaDOS Packet Interface Specification'' article on page II-5 of
Amiga Mail.

The ACTION_DISK_INFO packet is peculiar because its function changes
depending on the handler. When sent to a file system handler, it
returns information about its media. The dos.library function Info()
uses this packet. The ACTION_DISK_INFO packet has a different
meaning to a console handler. When a console handler receives this
packet, it returns a pointer to the window of the open console file
handle. When an application needs this pointer, the only way to get
it as of Release 3.0 is to send the console handler this packet.

There are two commonly used packets that an application could not
call through a dos.library function under 1.3 that now have
corresponding dos.library functions. The packet to rename a disk,
ACTION_RENAME_DISK, is now available through the dos.library function
Relabel(). Also the ACTION_SCREEN_MODE packet acquired a dos.library
function, SetMode().

About the Examples

Two examples accompany this article. The first, CompareIO.c, uses
packet level I/O to copy the standard input channel to the standard
output channel (as set up by the standard startup code, c.o).
CompareIO uses both synchronous and asynchronous I/O to perform the
copy and reports the time it takes to do each. The other example,
InOutCTRL-C.c, also uses packets to copy the standard input channel
to the standard output channel, but it only uses asynchronous I/O.
The second example does a better job checking for a user break.

 CompareIO.c
 InOutCTRL-C.c