

Devices_Manual

COLLABORATORS

	<i>TITLE :</i> Devices_Manual		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY		July 18, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1	Devices_Manual	1
1.1	Amiga® RKM Devices: 3 Clipboard Device	1
1.2	3 Clipboard Device / Clipboard Device Commands and Functions	2
1.3	3 Clipboard Device / Device Interface	3
1.4	3 / Device Interface / Opening The Clipboard Device	3
1.5	3 / Device Interface / Clipboard Data	4
1.6	3 / Device Interface / Multiple Clips	5
1.7	3 / Device Interface / Writing To The Clipboard Device	5
1.8	3 / Device Interface / Updating The Clipboard Device	7
1.9	3 / Device Interface / Clipboard Messages	7
1.10	3 / Device Interface / Reading From The Clipboard Device	8
1.11	3 / Device Interface / Closing The Clipboard Device	9
1.12	3 Clipboard Device / Monitoring Clipboard Changes	9
1.13	3 / Monitoring Clipboard Changes / Caveats For CBD_CHANGEHOOK	10
1.14	3 Clipboard Device / Additional Information on the Clipboard Device	10

Chapter 1

Devices_Manual

1.1 Amiga® RKM Devices: 3 Clipboard Device

The clipboard device allows the exchange of data dynamically between one application and another. It is responsible for caching data that has been "cut" and providing data to "paste" in an application. A special "post" mode allows an application to inform the clipboard device that the application has data available. The clipboard device will request this data only if the data is actually needed. The clipboard will cache the data in RAM and will automatically spool the data to disk if necessary.

The clipboard device is implemented as an Exec-style device, and supports random access reads and writes on data within the clipboard. All data in the clipboard must be in IFF format. A new library, IFFParse Library, has been added to the Amiga libraries. The routines in iffparse.library can and should be used for reading and writing data to the clipboard. This chapter contains a brief discussion of IFF as it relates to the clipboard (for more details see Appendix A).

NEW CLIPBOARD FEATURES FOR VERSION 2.0

Feature	Description
-----	-----
CBD_CHANGEHOOK	Device Command

Compatibility Warning:

The new features for the 2.0 clipboard device are not backwards compatible.

Clipboard Device Commands and Functions
Device Interface
Monitoring Clipboard Changes
Example Clipboard Programs
Support Functions Called from Example Programs
Include File for the Example Programs
Additional Information on the Clipboard Device

1.2 3 Clipboard Device / Clipboard Device Commands and Functions

Command -----	Command Operation -----
CBD_CHANGEHOOK	Specify a hook to be called when the data on the clipboard has changed (V36).
CBD_CURRENTREADID	Return the Clip ID of the current clip to read. This is used to determine if a clip posting is still the latest cut.
CBD_CURRENTWRITEID	Return the Clip ID of the latest clip written. This is used to determine if the clip posting data is obsolete.
CBD_POST	Post the availability of clip data.
CMD_READ	Read data from the clipboard for a paste. Data can be read from anywhere in the clipboard by specifying an offset >0 in the I/O request.
CMD_UPDATE	Indicate that the data provided with a write command is complete and available for subsequent read/pastes.
CMD_WRITE	Write data to the clipboard as a cut.

Exec Functions as Used in This Chapter

CloseDevice()	Relinquish use of the clipboard device. All requests must be complete before closing.
DoIO()	Initiate a command and wait for completion (synchronous request).
GetMsg()	Get next message from a message port.
OpenDevice()	Obtain use of the clipboard device.
SendIO()	Initiate a command and return immediately (asynchronous request).

Exec Support Functions as Used in This Chapter

CreateExtIO()	Create an I/O request structure of type IOClipReq. This structure will be used to communicate commands to the clipboard device.
CreatePort()	Create a signal message port for reply messages from the clipboard device. Exec will signal a task when a message arrives at the port.
DeleteExtIO()	Delete an I/O request structure created by CreateExtIO().

DeletePort() Delete the message port created by CreatePort().

1.3 3 Clipboard Device / Device Interface

The clipboard device operates like the other Amiga devices. To use it, you must first open the clipboard device, then send I/O requests to it, and then close it when finished. See "Introduction to Amiga System Devices" chapter for general information on device usage.

```
struct IOClipReq
{
    struct Message io_Message;
    struct Device *io_Device;      /* device node pointer */
    struct Unit *io_Unit;          /* unit (driver private) */
    UWORD io_Command;             /* device command */
    UBYTE io_Flags;               /* including QUICK and SATISFY */
    BYTE io_Error;                /* error or warning num */
    ULONG io_Actual;              /* number of bytes transferred */
    ULONG io_Length;              /* number of bytes requested */
    STRPTR io_Data;               /* either clip stream or post port */
    ULONG io_Offset;              /* offset in clip stream */
    LONG io_ClipID;               /* ordinal clip identifier */
};
```

See the include file devices/clipboard.h for the complete structure definition.

The clipboard device I/O request, IOClipReq, looks like a standard IORequest structure except for the addition of the io_ClipID field, which is used by the device to identify clips. It must be set to zero by the application for a post or an initial write or read, but preserved for subsequent writes or reads, as the clipboard device uses this field internally for bookkeeping purposes.

Opening The Clipboard Device	Updating The Clipboard Device
Clipboard Data	Clipboard Messages
Multiple Clips	Reading From The Clipboard Device
Writing To The Clipboard Device	Closing The Clipboard Device

1.4 3 / Device Interface / Opening The Clipboard Device

Three primary steps are required to open the clipboard device:

- * Create a message port using CreatePort(). Reply messages from the device must be directed to a message port.
 - * Create an extended I/O request structure of type IOClipReq using CreateExtIO().
 - * Open the clipboard device. Call OpenDevice(), passing the IOClipReq.
-

```

struct MsgPort *ClipMP;          /* pointer to message port */
struct IOClipReq *ClipIO;        /* pointer to IORequest */

if (ClipMP=CreatePort(0L,0L) )
{
    if (ClipIO=(struct IOClipReq *)
        CreateExtIO(ClipMP,sizeof(struct IOClipReq)))
    {
        if (OpenDevice("clipboard.device",0L,ClipIO,0))
            printf("clipboard.device did not open\n");
        else
        {
            ... do device processing
        }
    }
    else
        printf("Error: Could not create IORequest\n");
}
else
    printf("Error: Could not create message port\n");

```

1.5 3 / Device Interface / Clipboard Data

Data on the clipboard resides in one of three places. When an application posts a cut, the data resides in the private memory space of that application. When an application writes to the clipboard, either of its own volition or in response to a message from the clipboard requesting that it satisfy a post, the data is copied to the clipboard which is either memory or a special disk file. When the clipboard is not open, the data resides in the special disk file located in the directory specified by the CLIPS: logical AmigaDOS assign.

Data on the clipboard is self-identifying. It must be a correct IFF (Interchange File Format) file; the rest of this section refers to IFF concepts. See the Appendix A in this manual for a complete description of IFF. If the top-level chunk is of type CAT with an identifier of CLIP, that indicates that the contained chunks are different representations of the same data, in decreasing order of preference on the part of the producer of the clip. Any other data is as defined elsewhere (probably a single representation of the cut data produced by an application).

The IFFParse.Library also contains functions which simplify reading and writing of IFF data to the clipboard device. See the "IFFParse Library" chapter of the Amiga ROM Kernel Reference Manual: Libraries for more information.

A clipboard tool, which is an application that allows a Workbench user to view a clip, should understand the text (FTXT) and graphics (ILBM) form types. Applications using the clipboard to export data should include at least one of these types in a CAT CLIP so that their data can be represented on the clipboard in some form for user feedback.

You should not, in any way, rely on the specifics of how files in CLIPS: are handled or named. The only proper way to read or write clipboard data is via the clipboard device.

Play Nice!

Keep in mind that while your task is reading from or writing to a clipboard unit, other tasks cannot. Therefore, it is important to be fast. If possible, make a copy of the clipboard data in RAM instead of processing it while the read or write is in progress.

1.6 3 / Device Interface / Multiple Clips

The clipboard supports multiple clips, i.e., the clipboard device can contain more than one distinct piece of data. This is not to be confused with the IFF CAT CLIP, which allows for different representation of the same data.

The multiple clips are implemented as different units in the clipboard device. The unit is specified at `OpenDevice()` time.

```
struct IOClipReq *ClipIO;
LONG unit;
```

```
OpenDevice("clipboard.device", unit, ClipIO, 0);
```

By default, applications should use clipboard unit 0. However, it is recommended that each application provide a mechanism for selecting the unit number which will be used when the clipboard is opened. This will allow the user to create a convention for storing different types of data in the clipboard. Applications should never write to clipboard unit 0 unless the user requests it (e.g., selecting COPY or CUT within an application).

Clipboard units 1-255 can be used by the more advanced user for:

- * Sharing data between applications within an ARexx Script.
- * Customizing applications to store different kinds of data in different clipboard units.
- * Customizing an application to use multiple cut/copy/paste buffers.
- * Specialized utilities which might display and/or automatically modify the contents of a clipboard unit.

All applications which provide CUT, COPY and PASTE capabilities, should, at a minimum, provide support for clipboard unit 0.

1.7 3 / Device Interface / Writing To The Clipboard Device

You write to the clipboard device by passing an `IOClipReq` to the device with `CMD_WRITE` set in `io_Command`, the number of bytes to be written set in `io_Length` and the address of the write buffer set in `io_Data`.


```
ClipIO->io_Data = (char *) data;
ClipIO->io_Length = 4L;
ClipIO->io_Command = CMD_WRITE;
```

An initial write should set `io_Offset` to zero. Each time a write is done, the device will increment `io_Offset` by the length of the write.

As previously stated, the data you write to the clipboard must be in IFF format. This requires a certain amount of preparation prior to actually writing the data if it is not already in IFF format. A brief explanation of the IFF format will be helpful in this regard.

For our purposes, we will limit our discussion to a simple formatted text (FTXT) IFF file. An FTXT file looks like:

```
FORM <length of succeeding bytes>
FTXT
CHRS <length of succeeding bytes>
      <data bytes>
      <pad byte of zero if the preceding chunk has odd length>
```

Note: Uppercase characters above are literals.

Based on the above file format, a hex dump of an IFF FTXT file containing the string "Enterprise" would look like:

```
0000  464F524D  FORM
0004  00000016  (length of FTXT, CHRS, 0xA and data)
0008  46545854  FTXT
000C  43485253  CHRS
0010  0000000A  (length of Enterprise)
0014  456E7465  Ente
0018  72707269  rpri
001C  7365      se
```

A code fragment for doing this:

```
LONG slen = strlen ("Enterprise");
BOOL odd = (slen & 1);      /* pad byte flag */

/* set length depending on whether string is odd or even length */
LONG length = (odd) ? slen + 1 : slen;

/* Reset the clip id */
ClipIO->io_ClipID = 0;
ClipIO->io_Offset = 0;

error = writeLong ((LONG *) "FORM"); /* "FORM" */

/* add 12 bytes for FTXT,CHRS & length byte to string length */
length += 12;
error = writeLong (&length);
error = writeLong ((LONG *) "FTXT"); /* "FTXT" for example */
error = writeLong ((LONG *) "CHRS"); /* "CHRS" for example */
error = writeLong (&slen);          /* # (length of string) */

ClipIO->io_Command = CMD_WRITE;
```

```

ClipIO->io_Data = (char *) string;
ClipIO->io_Length = slen;                /* length of string */
error = (LONG) DoIO (clipIO);           /* text string */

LONG writeLong (LONG * ldata)
{
    ClipIO->io_Command = CMD_WRITE;
    ClipIO->io_Data = (char *) ldata;
    ClipIO->io_Length = 4L;
    return ( (LONG) DoIO (clipIO) );
}

```

The fragment above does no error checking because it's a fragment. You should always error check. See the example programs at the end of this chapter for the proper method of error checking.

Iffparse That Data!

Keep in mind that the functions in the `iffparse.library` can be used to write data to the clipboard. See the "IFFParse Library" chapter of the Amiga ROM Kernel Reference Manual: Libraries for more information.

1.8 3 / Device Interface / Updating The Clipboard Device

When the final write is done, an update command must be sent to the device to indicate that the writing is complete and the data is available. You update the clipboard device by passing an `IOClipReq` to the device with `CMD_UPDATE` set in `io_Command`.

```

ClipIO->io_Command = CMD_UPDATE;
DoIO(ClipIO);

```

1.9 3 / Device Interface / Clipboard Messages

When an application performs a post, it must specify a message port for the clipboard to send a message to if it needs the application to satisfy the post with a write called the `SatisfyMsg`.

```

struct SatisfyMsg
{
    struct Message sm_Message; /* the length will be 6 */
    UWORD    sm_Unit;          /* 0 for the primary clip unit */
    LONG     sm_ClipID;        /* the clip identifier of the post */
}

```

This structure is defined in the include file `devices/clipboard.h`.

If the application wishes to determine if a post it has recently performed is still the current clip, it should compare the `io_ClipID` found in the post request upon return with that returned by the `CBD_CURRENTREADID` command.

If an application has a pending post and wishes to determine if it should satisfy it (for example, before it exits), it should compare the `io_ClipID` of the post I/O request with that of the `CBD_CURRENTWRITEID` command. If the application receives a satisfy message from the clipboard device (format described below), it must immediately perform the write with the `io_ClipID` of the post. The satisfy message from the clipboard may be removed from the application message port by the clipboard device at any time (because it is re-used by the clipboard device). It is not dangerous to spuriously satisfy a post, however, because it is identified by the `io_ClipID`.

The cut data is provided to the clipboard device via either a write or a post of the cut data. The write command accepts the data immediately and copies it onto the clipboard. The post command allows an application to inform the clipboard of a cut, but defers the write until the data is actually required for a paste.

In the preceding discussion, references to the read and write commands of the clipboard device actually refer to a sequence of read or write commands, where the clip data is acquired and provided in pieces instead of all at once.

The clipboard has an end-of-clip concept that is analogous to end-of-file for both read and write. The read end-of-file must be triggered by the user of the clipboard in order for the clipboard to move on to service another application's requests, and consists of reading data past the end of file. The write end-of-file is indicated by use of the update command, which indicates to the clipboard that the previous write commands are completed.

1.10 3 / Device Interface / Reading From The Clipboard Device

You read from the clipboard device by passing an `IOClipReq` to the device with `CMD_READ` set in `io_Command`, the number of bytes to be read set in `io_Length` and the address of the read buffer set in `io_Data`.

```
ClipIO->io_Command = CMD_READ;
ClipIO->io_Data = (char *) read_data;
ClipIO->io_Length = 20L;
```

`io_Offset` must be set to zero for the first read of a paste sequence. An `io_Actual` that is less than the `io_Length` indicates that all the data has been read. After all the data has been read, a subsequent read must be performed (one whose `io_Actual` returns zero) to indicate to the clipboard device that all the data has been read. This allows random access of the clip while reading. Providing only valid reads are performed, your program can seek/read anywhere within the clip by setting the `io_Offset` field of the I/O request appropriately.

Tell The Clipboard You Are Finished Reading.

Your application must perform an extra read (one whose `io_Actual` returns zero) to indicate to the clipboard device that all data has been read, if `io_Actual` is not already zero.

The data you read from the clipboard will be in IFF format. Conversion from IFF may be necessary depending on your application.

Iffparse That Data!

Keep in mind that the functions in the `iffparse.library` can be used to read data from the clipboard. See the "IFFParse Library" chapter of the Amiga ROM Kernel Reference Manual: Libraries for more information.

1.11 3 / Device Interface / Closing The Clipboard Device

Each `OpenDevice()` must eventually be matched by a call to `CloseDevice()`.

```
CloseDevice(ClipIO);
```

When the last task closes a clipboard unit with `CloseDevice()`, the contents of the unit may be copied to a disk file in `CLIPS:` so that the clipboard device can be expunged.

1.12 3 Clipboard Device / Monitoring Clipboard Changes

Some applications require notification of changes to data on the clipboard. Typically, these applications will need to do some processing when this occurs. You can set up such an environment through the `CBD_CHANGEHOOK` command. `CBD_CHANGEHOOK` allows you to specify a hook to be called when the data on the clipboard changes.

For example, a show clipboard utility would need to know when the data on the clipboard is changed so that it can display the new data. The hook it would specify would read the new clipboard data and display it for the user.

You specify a hook for the clipboard device by initializing a Hook structure and then passing an `IOClipReq` to the device with `CBD_CHANGEHOOK` set in `io_Command`, 1 set in `io_Length`, and the address of the Hook structure set in `io_Data`.

```
ULONG HookEntry ();                /* Declare the hook assembly function */
struct IOClipReq *ClipIO;          /* Declare the IOClipReq */
struct Hook *ClipHook;             /* Declare the Hook */

/* Prepare the hook */
ClipHook->h_Entry = HookEntry; /* C intrfce in asmbly rout. HookEntry*/
ClipHook->h_SubEntry = HookFunc; /* Call function when Hook activated */
ClipHook->h_Data = FindTask(NULL); /* Set pointer to current task */

ClipIO->io_Data = (char *) ClipHook; /* Point to hook struct */
ClipIO->io_Length = 1;                /* Add hook to clipboard */
ClipIO->io_Command = CBD_CHANGEHOOK;
```

```
DoIO(clipIO);
```

The above code fragment assumes that an assembly language routine `HookEntry()` has been coded:

```
; entry interface for C code
_HookEntry:
    move.l    a1,-(sp)        ; push message packet pointer
    move.l    a2,-(sp)        ; push object pointer
    move.l    a0,-(sp)        ; push hook pointer
    move.l    h_SubEntry(a0),a0 ; fetch C entry point ...
    jsr       (a0)            ; ... and call it
    lea       12(sp),sp       ; fix stack
    rts
```

It also assumes that the function `HookFunc()` has been coded. One of the example programs at the end of this chapter has hook processing in it. See the include file `utility/hooks.h` and *The Amiga ROM Kernel Reference Manual: Libraries* for further information on hooks.

You remove a hook by passing an `IOClipReq` to the device with the address of the `Hook` structure set in `io_Data`, 0 set in `io_Length` and `CBD_CHANGEHOOK` set in `io_Command`.

```
ClipIO->io_Data = (char *) ClipHook; /* point to hook struct */
ClipIO->io_Length = 0;                /* Remove hook from clipboard */
ClipIO->io_Command = CBD_CHANGEHOOK;
(DoIO (clipIO))
```

You must remove the hook or it will continue indefinitely.

Caveats For `CBD_CHANGEHOOK`

1.13 3 / Monitoring Clipboard Changes / Caveats For `CBD_CHANGEHOOK`

- * `CBD_CHANGEHOOK` should only be used by a special application, such as a clipboard viewing program. Most applications can check the contents of the clipboard when, and if, the user requests a paste.
- * Do not add system overhead by blindly reading and parsing the clipboard everytime a user copies data to it. If all applications did this, the system could become intolerably slow whenever an application wrote to the clipboard. Only read and parse when it is necessary.

1.14 3 Clipboard Device / Additional Information on the Clipboard Device

Additional programming information on the clipboard device can be found in the include files for the clipboard device, `iffparse` library and `utility` library, and the Autodocs for all three. They are contained in the *Amiga ROM Kernel Reference Manual: Includes and Autodocs*.

Clipboard Device Information

INCLUDES	devices/clipboard.h devices/clipboard.i libraries/iffparse.h libraries/iffparse.i utility/hooks.h utility/hooks.i
AUTODOCS	clipboard.doc iffparse.doc utility.doc
