

Devices

COLLABORATORS

	<i>TITLE :</i> Devices		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY		July 23, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1	Devices	1
1.1	Amiga® RKM Devices: 1 Introduction to Amiga System Devices	1
1.2	1 Introduction to Amiga System Devices / What is a Device?	1
1.3	1 Introduction to Amiga System Devices / Accessing a Device	2
1.4	1 Introduction to Amiga System Devices / Using a Device	4
1.5	1 / Using a Device / Amiga Exec Commands	5
1.6	1 / Using a Device / Amiga System Device Command Prefixes and Examples	5
1.7	1 Introduction Amiga System Devices / Synchronous vs. Asynchronous Requests	5
1.8	1 Introduction to Amiga System Devices / I/O Request Completion	7
1.9	1 Introduction to Amiga System Devices / Ending Device Access	7
1.10	1 Introduction to Amiga System Devices / Devices With Functions	9
1.11	1 Introduction to Amiga System Devices / Example Device Programs	10

Chapter 1

Devices

1.1 Amiga® RKM Devices: 1 Introduction to Amiga System Devices

The Amiga system devices are software engines that provide access to the Amiga hardware. Through these devices, a programmer can operate a modem, spin a disk drive motor, time an event, speak to a user and blast a trumpet sound in beautiful, living stereo. Yet, for all that variety, the programmer uses each device in the same basic manner.

Amiga System Devices

Audio	Controls the use of the audio hardware
Clipboard	Manages the cutting and pasting of common data blocks
Console	Provides the text-oriented user interface.
Gameport	Controls the two mouse/joystick ports.
Input	Processes input from the gameport and keyboard devices.
Keyboard	Controls the keyboard.
Narrator	Produces the Amiga synthesized speech.
Parallel	Controls the parallel port.
Printer	Converts a standard set of printer control codes to printer specific codes.
SCSI	Controls the Small Computer Standard Interface hardware.
Serial	Controls the serial port.
Timer	Provides timing functions to measure time intervals and send interrupts.
Trackdisk	Controls the Amiga floppy disk drives.

What is a Device?	I/O Request Completion
Accessing a Device	Ending Device Access
Using a Device	Devices With Functions
Synchronous vs. Asynchronous Requests	Example Device Programs

1.2 1 Introduction to Amiga System Devices / What is a Device?

An Amiga device is a software module that accepts commands and data and performs I/O operations based on the commands it receives. In most cases, a device interacts with either internal or external hardware. Generally, an Amiga device runs as a separate task which is capable of processing

your commands while your application attends to other things.

Device I/O is based on the Exec messaging system. The philosophy behind the devices is that I/O operations should be consistent and uniform. You print a file in the same manner as you play an audio sample, i.e., you send the device in question a WRITE command and the address of the buffer holding the data you wish to write.

The result is that the interface presented to the programmer is essentially device independent and accessible from any computer language. This greatly expands the power the Amiga computer brings to the programmer and, ultimately, to the user.

Devices support two types of commands: Exec standard commands like READ and WRITE, and device specific commands like the trackdisk device MOTOR command which controls the floppy drive motor. The Exec standard commands are supported by most Amiga devices. You should keep in mind, however, that supporting standard commands does not mean that all devices execute them in exactly the same manner.

This manual contains a chapter about each of the Amiga devices. The chapters cover how you use a device and the commands it supports. In addition, the Amiga ROM Kernel Reference Manual: Includes and Autodocs contains expanded explanations of the commands and the include files for each device, and the Amiga ROM Kernel Reference Manual: Libraries contains chapters on Exec. The command explanations list the data, flags, and other information required by a device to execute a command. The Exec chapters provide detailed discussions of its operation. Both are very useful manuals to have on your desk when you are programming the devices.

1.3 1 Introduction to Amiga System Devices / Accessing a Device

Accessing a device requires obtaining a message port, allocating memory for a specialized message packet called an I/O request, setting a pointer to the message port in the I/O request, and finally, establishing the link to the device itself by opening it. An example of how to do this will be provided later in this chapter.

The message port is used by the device to return messages to you. A message port is obtained by calling the `CreateMsgPort()` or `CreatePort()` function. You must delete the message port when you are finished by calling the `DeleteMsgPort()` or `DeletePort()` function.

For pre-V36 versions of the operating system (before Release 2.0), use the `amiga.lib` functions `CreatePort()` and `DeletePort()`; for V36 and higher, use the Exec functions `CreateMsgPort()` and `DeleteMsgPort()`. `CreatePort()` and `DeletePort()` are upward compatible, you can use them with V36/V37; `CreateMsgPort()` and `DeleteMsgPort()` are not backward compatible, however.

The I/O request is used to send commands and data from your application to the device. The I/O request consists of fields used to hold the command you wish to execute and any parameters it requires. You set up the fields with the appropriate information and send it to the device by using Exec I/O functions.

At least four methods exist for creating an I/O request:

- * Declaring it as a structure. The memory required will be allocated at compile time.
- * Declaring it as a pointer and calling the AllocMem() function. You will have to call the FreeMem() function to release the memory when you are done.
- * Declaring it as a pointer and calling the CreateExtIO() function. This function not only allocates the memory for the request, it also puts the message port in the I/O request. You will have to call the DeleteExtIO() function to delete the I/O request when you are done. This is the pre-V36 method (used in 1.3 and earlier versions of the operating system), but is upward compatible.
- * Declaring it as a pointer and calling the CreateIORequest() function. This function not only allocates the memory for the request, it also puts the message port in the I/O request. You will have to call the DeleteIORequest() function to delete the I/O request when you are done. This is the V36/V37 method; it is not backwards compatible.

The message port pointer in the I/O request tells the device where to respond with messages for your application. You must set a pointer to the message port in the I/O request if you declare it as a structure or allocate memory for it using AllocMem().

The device is opened by calling the OpenDevice() function. In addition to establishing the link to the device, OpenDevice() also initializes fields in the I/O request. OpenDevice() has this format:

```
return = OpenDevice(device_name,unit_number,
                    (struct IORequest *)IORequest,flags)
```

where:

- * device_name is one of the following NULL-terminated strings for system devices:

audio.device	keyboard.device	serial.device
clipboard.device	narrator.device	timer.device
console.device	parallel.device	trackdisk.device
gameport.device	printer.device	input.device
scsi.device		
- * unit_number refers to one of the logical units of the device. Devices with one unit always use unit 0. Multiple unit devices like the trackdisk device and the timer device use the different units for specific purposes. The device chapters discuss the units in detail.
- * IORequest is the structure discussed above. Some of the devices have their own I/O requests defined in their include files and others use standard I/O requests, (IOStdReq). The device chapters list the I/O request that each device requires.
- * flags are bits set to indicate options for some of the devices. This field is set to zero for devices which don't accept options when they

are opened. The device chapters and autodocs list the flags values and uses.

- * return is an indication of whether the `OpenDevice()` was successful with zero indicating success. Never assume that a device will successfully open. Check the return value and act accordingly.

Zero Equals Success for `OpenDevice()`.

Unlike most Amiga system functions, `OpenDevice()` returns zero for success and a device-specific error value for failure.

1.4 1 Introduction to Amiga System Devices / Using a Device

Once a device has been opened, you use it by passing the I/O request to it. When the device processes the I/O request, it acts on the information the I/O request contains and returns a reply message, i.e., the I/O request, to the reply port specified in the I/O request when it is finished. The I/O request is passed to a device using one of three functions, `DoIO()`, `SendIO()` and `BeginIO()`. They take only one argument: the I/O request you wish to pass to the device.

- * `DoIO()` is a synchronous function. It will not return until the device has responded to the I/O request.
- * `SendIO()` is an asynchronous function. It can return immediately, but the I/O operation it initiates may take a short or long time. Using `SendIO()` requires you to monitor the message port for a return message from the device. In addition, some devices do not actually respond asynchronously even though you called `SendIO()`; they will return when the I/O operation is finished.
- * `BeginIO()` is commonly used to control the quick I/O bit when sending an I/O request to a device. When the quick I/O flag (`IOF_QUICK`) is set in the I/O request, a device is allowed to take certain shortcuts in performing and completing a request. If the request can complete immediately, the device will not return a reply message and the quick I/O flag will remain set. If the request cannot be completed immediately, the `QUICK_IO` flag will be clear. `DoIO()` normally requests quick I/O; `SendIO()` does not.

`DoIO()` and `SendIO()` are most commonly used.

An I/O request typically has three fields set for every command sent to a device. You set the command itself in the `io_Command` field, a pointer to the data for the command in the `io_Data` field, and the length of the data in the `io_Length` field.

```
SerialIO->IOSer.io_Length = sizeof(ReadBuffer);
SerialIO->IOSer.io_Data = ReadBuffer;
SerialIO->IOSer.io_Command = CMD_READ;
SendIO((struct IORequest *)SerialIO);
```

Commands consist of two parts - a prefix and the command word separated by

an underscore, all in upper case. The prefix indicates whether the command is an Exec or device specific command. All Exec commands have CMD as the prefix. They are defined in the include file exec/io.h.

Amiga Exec Commands
Amiga System Device Command Prefixes and Examples

1.5 1 / Using a Device /Amiga Exec Commands

CMD_CLEAR	CMD_READ	CMD_STOP
CMD_FLUSH	CMD_RESET	CMD_WRITE
CMD_INVALID	CMD_START	CMD_UPDATE

You should not assume that a device supports all Exec commands. Always check the documentation before attempting to use one of them.

Device specific command prefixes vary with the device.

1.6 1 / Using a Device / Amiga System Device Command Prefixes and Examples

Device	Prefix	Example
-----	-----	-----
Audio	ADCMD	ADCMD_ALLOCATE
Clipboard	CBD	CBD_POST
Console	CD	CD_ASKKEYMAP
Gameport	GPD	GPD_SETCTYPE
Input	IND	IND_SETMPORT
Keyboard	KBD	KBD_READMATRIX
Narrator	no device specific commands	
Parallel	PDCMD	PDCMD_QUERY
Printer	PRD	PRD_PRTCOMMAND
SCSI	HD	HD_SCSICMD
Serial	SDCMD	SDCMD_BREAK
Timer	TR	TR_ADDREQUEST
Trackdisk	TD and ETD	TD_MOTOR/ETD_MOTOR

Each device maintains its own I/O request queue. When a device receives an I/O request, it either processes the request immediately or puts it in the queue because one is already being processed. After an I/O request is completely processed, the device checks its queue and if it finds another I/O request, begins to process that request.

1.7 1 Introduction Amiga System Devices / Synchronous vs. Asynchronous Requests

As stated above, you can send I/O requests to a device synchronously or asynchronously. The choice of which to use is largely a function of your application.

Synchronous requests use the `DoIO()` function. `DoIO()` will not return control to your application until the I/O request has been satisfied by the device. The advantage of this is that you don't have to monitor the message port for the device reply because `DoIO()` takes care of all the message handling. The disadvantage is that your application will be tied up while the I/O request is being processed, and should the request not complete for some reason, `DoIO()` will not return and your application will hang.

Asynchronous requests use the `SendIO()` and `BeginIO()` functions. Both return to your application almost immediately after you call them. This allows you to do other operations, including sending more I/O requests to the device.

Do Not Touch!

When you use `SendIO()` or `BeginIO()`, the I/O request you pass to the device and any associated data buffers should be considered read-only. Once you send it to the device, you must not modify it in any way until you receive the reply message from the device or abort the request (though you must still wait for a reply). Any exceptions to this rule are documented in the autodoc for the device.

Sending multiple asynchronous I/O requests to a device can be tricky because devices require them to be unique and initialized. This means you can't use an I/O request that's still in the queue, but you need the fields which were initialized in it when you opened the device. The solution is to copy the initialized I/O request to another I/O request(s) before sending anything to the device.

Regardless of what you do while you are waiting for an asynchronous I/O request to return, you need to have some mechanism for knowing when the request has been done. There are two basic methods for doing this.

The first involves putting your application into a wait state until the device returns the I/O request to the message port of your application. You can use the `WaitIO()`, `Wait()` or `WaitPort()` function to wait for the return of the I/O request.

`WaitIO()` not only waits for the return of the I/O request, it also takes care of all the message handling functions. This is very convenient, but you can pay for this convenience: your application will hang in the unlikely event that the I/O request does not return.

`Wait()` waits for a signal to be sent to the message port. It will awaken your task when the signal arrives, but you are responsible for all of the message handling.

`WaitPort()` waits for the message port to be non-empty. It returns a pointer to the message in the port, but you are responsible for all of the message handling.

The second method to detect when the request is complete involves using the `CheckIO()` function. `CheckIO()` takes an I/O request as its argument and returns an indication of whether or not it has been completed. When `CheckIO()` returns the completed indication, you will still have to remove

the I/O request from the message port.

1.8 1 Introduction to Amiga System Devices / I/O Request Completion

A device will set the `io_Error` field of the I/O request to indicate the success or failure of an operation. The indication will be either zero for success or a non-zero error code for failure. There are two types of error codes: Exec I/O and device specific. Exec I/O errors are defined in the include file `exec/errors.h`; device specific errors are defined in the include file for each device. You should always check that the operation you requested was successful.

The exact method for checking `io_Error` can depend on whether you use `DoIO()` or `SendIO()`. In both cases, `io_Error` will be set when the I/O request is returned, but in the case of `DoIO()`, the `DoIO()` function itself returns the same value as `io_Error`.

This gives you the option of checking the function return value:

```
SerialIO->IOSer.io_Length  = sizeof(ReadBuffer);
SerialIO->IOSer.io_Data    = ReadBuffer;
SerialIO->IOSer.io_Command = CMD_READ;
if (DoIO((struct IORequest *)SerialIO);
    printf("Read failed.  Error: %ld\n",SerialIO->IOSer.io_Error);
```

Or you can check `io_Error` directly:

```
SerialIO->IOSer.io_Length  = sizeof(ReadBuffer);
SerialIO->IOSer.io_Data    = ReadBuffer;
SerialIO->IOSer.io_Command = CMD_READ;
DoIO((struct IORequest *)SerialIO);

if (SerialIO->IOSer.io_Error)
    printf("Read failed.  Error: %ld\n",SerialIO->IOSer.io_Error);
```

Keep in mind that checking `io_Error` is the only way that I/O requests sent by `SendIO()` can be checked.

Testing for a failed I/O request is a minimum step, what you do beyond that depends on your application. In some instances, you may decide to resend the I/O request, and in others, you may decide to stop your application. One thing you'll almost always want to do is to inform the user that an error has occurred.

Exiting The Correct Way.

If you decide that you must prematurely end your application, you should deallocate, release, give back and let go of everything you took to run the application. In other words, you should exit gracefully.

1.9 1 Introduction to Amiga System Devices / Ending Device Access

You end device access by reversing the steps you took to access it. This means you close the device, deallocate the I/O request memory and delete the message port. In that order!

Closing a device is how you tell Exec that you are finished using a device and any associated resources. This can result in housecleaning being performed by the device. However, before you close a device, you might have to do some housecleaning of your own.

A device is closed by calling the `CloseDevice()` function. The `CloseDevice()` function does not return a value. It has this format:

```
CloseDevice(IORequest)
```

where `IORequest` is the I/O request used to open the device.

You should not close a device while there are outstanding I/O requests, otherwise you can cause major and minor problems. Let's begin with the minor problem: memory. If an I/O request is outstanding at the time you close a device, you won't be able to reclaim the memory you allocated for it.

The major problem: the device will try to respond to the I/O request. If the device tries to respond to an I/O request, and you've deleted the message port (which is covered below), you will probably crash the system.

One solution would be to wait until all I/O requests you sent to the device return. This is not always practical if you've sent a few requests and the user wants to exit the application immediately.

In that case, the only solution is to abort and remove any outstanding I/O requests. You do this with the functions `AbortIO()` and `WaitIO()`. They must be used together for cleaning up. `AbortIO()` will abort an I/O request, but will not prevent a reply message from being sent to the application requesting the abort. `WaitIO()` will wait for an I/O request to complete and remove it from the message port. This is why they must be used together.

```
Be Careful With AbortIO().
```

```
-----
```

```
Do not AbortIO() an I/O request which has not been sent to a device.  
If you do, you may crash the system.
```

After the device is closed, you must deallocate the I/O request memory. The exact method you use depends on how you allocated the memory in the first place. For `AllocMem()` you call `FreeMem()`, for `CreateExtIO()` you call `DeleteExtIO()`, and for `CreateIORequest()` you call `DeleteIORequest()`. If you allocated the I/O request memory at compile time, you naturally have nothing to free.

Finally, you must delete the message port you created. You delete the message port by calling `DeleteMsgPort()` if you used `CreateMsgPort()`, or `DeletePort()` if you used `CreatePort()`.

Here is the checklist for gracefully exiting:

1. Abort any outstanding I/O requests with `AbortIO()`
2. Wait for the completion of any outstanding or aborted I/O requests with `WaitIO()`.
3. Close the device with `CloseDevice()`.
4. Release the I/O request memory with either `DeleteIORequest()`, `DeleteExtIO()` or `FreeMem()` (as appropriate).
5. Delete the message port with `DeleteMsgPort()` or `DeletePort()`.

1.10 1 Introduction to Amiga System Devices / Devices With Functions

Some devices, in addition to their commands, provide functions which can be directly called by applications. These functions are documented in the device specific FD files and Autodocs of the Amiga ROM Kernel Reference Manual: Includes and Autodocs and the device chapters of this manual.

Devices with functions behave much like Amiga libraries, i.e., you set up a base address pointer and call the functions as offsets from the pointer. (See the "Exec Libraries" chapter of the Amiga ROM Kernel Reference Manual: Libraries.)

The procedure for accessing a device's functions is as follows:

- * Declare the device base address variable in the global data area. The name of the base address can be found in the device's FD file.
- * Create a message port using one of the previously discussed methods if you haven't already done so.
- * Create an I/O request using one of the previously discussed methods if you haven't already done so. Remember to set the message port pointer in the I/O request if necessary.
- * Call `OpenDevice()`, passing the I/O request if you haven't already done so. When you do this, the device returns a pointer to its base address in the `io_Device` field of the I/O request structure. Consult the include file for the structure you are using to determine the full name of the `io_Device` field. The base address is only valid while the device is open.
- * Set the device base address variable to the pointer returned in the `io_Device` field.

We will use the timer device to illustrate the above method. The name of the timer device base address is listed in its FD file as "TimerBase."

```
#include <devices/timer.h>

struct Library *TimerBase;    /* device base address pointer */

struct MsgPort *TimerMP;     /* message port pointer */
struct timerequest *TimerIO; /* I/O request pointer */
```

```
/* Create the message port */
if (TimerMP=CreatePort(NULL,NULL))
{
    /* Create the I/O request */
    if (TimerIO = (struct timerequest *)
        {
            CreateExtIO(TimerMP,sizeof(struct timerequest)))
            /* Open the timer device */
            if (!(OpenDevice(TIMERNAME,UNIT_MICROHZ,TimerIO,0)))
            {
                /* Set up pointer for timer functions */
                TimerBase = (struct Library *)TimerIO->tr_node.io_Device;

                ... use functions ...

                /* Close the timer device */
                CloseDevice(TimerIO);
            }

            /* Delete the I/O request */
            DeleteExtIO(TimerIO);
        }

    /* Delete the message port */
    DeletePort(TimerMP);
}
```

1.11 1 Introduction to Amiga System Devices / Example Device Programs

The following short programs are examples of how to use a device. Both send the serial device command SDCMD_QUERY to the serial device to determine the status of the serial device lines and registers. The first program is for pre-V36 versions of the operating system (before Release 2) and the second is for V36 and higher. You may use the pre-V36 version with V36 and higher, but you may not use the V36 version with older systems.

The programs differ in the way they create the message port and I/O request. The pre-V36 version uses the amiga.lib functions CreatePort() to create the message port and CreateExtIO() to create the I/O request; the V36 version uses the Exec functions CreateMsgPort() to create the message port and CreateIORequest() to create the I/O request. Those are the only differences.

Device Usage Example (Pre-V36)

Device Usage Example (Kickstart V36 And Up)