concurrent tasks and between clipboard buffers in the same task. This device follows the task-device message-port model presented in Chapter 1 but sometimes uses an additional message port.

Chapter 13 discusses the Timer device, which allows a task to control timing operations. The Timer device manages timing events and the signals that initiate specific task activities. Chapter 14 discusses the TrackDisk device, which allows a task to control disk operations. The TrackDisk device manages all aspects of the Amiga disk system.

The appendix presents programming statements that define the Exec-support library functions presented in Chapter 2. You can use these examples to develop your own C language functions. The index provides a useful guide to the information presented in this volume. In addition, you can refer to the Table of Contents to find device structures, functions, and commands.

Although this book is not specifically addressed to the subject of custom-built devices, you will find that the figures in this volume help you understand the form and concept of a device, and therefore help you formulate your own devices to add to the Amiga system.

A detailed glossary and a set of useful script files, as well as detailed explanations of other features of the Amiga C language device-programming system, have been added to the disk referenced at the end of Volume I. To obtain this disk, complete the order form in the back of Volume I and return it with your check or money order.

What the Amiga Can Do

The capabilities of the Amiga computer can best be understood by considering an example of an Amiga at work. Imagine that a new store in a shopping complex wanted to use a computer to present passersby with an eye-catching, entertaining presentation in order to entice them into their store. With an Amiga, the presentation could accomplish the following:

- Offer an attractive, five-minute video presentation with a voice-over narration and background music.
- Allow selection of another video sequence, voice-over narration, or music selection by presenting an easily understood range of keyboard choices.
- Allow alternative selections with a mouse to point to objects on the screen.
- Ask for responses (for example, as part of a survey).
- Provide a laser color printout as a memento of the experience.
- Allow the store owners to monitor responses through a modem in order to determine the customer's interests.
- Provide the store owners with a permanent record of customer-Amiga interactions.

This type of interaction is entirely within the reach of the Amiga computer. The Amiga can simultaneously produce stereo music from predigitalized audio tracks and a human-quality voice from predigitalized soundtracks, respond to input, print out information, and send information through the serial port to an attached modem.

To make all this simultaneous activity possible, the Amiga offers the following features:

- A large memory to accommodate the video and audio information, which can be produced and stored on disk in compressed form. The memory requirements of a typical five-minute audio-video presentation may require a sizable hard-disk drive.
- The ability to move information quickly from high RAM locations into lower RAM locations, where the hardware control chips can access that information and present it to the user.
- A user-friendly interface, with quick, quiet operation.
- A multitasking operating system, which allows many tasks to pass information among them and to signal each other of the information's arrival.
- The ability to accept input from a number of external sources (the keyboard, the mouse, gameport hardware, disk drives, and so on) simultaneously; to merge that data into the total input stream; and to act on those signals as requested, without interfering with a presentation.
- A responsive and fast system, which can work with many different categories of data at the same time.
- The ability to continuously and unobtrusively adapt to a real-time environment, where the sequence of events is not predetermined but can change as quickly as the user responds.

Amiga devices provide the means to program a complicated presentation such as this. They allow you to take full advantage of the Amiga's impressive capabilities.

The Device System

Seven devices—the Audio, Input, Console, Timer, Keyboard, Gameport, and TrackDisk devices—are ROM-resident. For the Amiga 1000, their internal routines, structures, and data are loaded into ROM when the system is first booted from the Kickstart disk. For the Amiga 500 and 2000, this information is already in ROM. Five devices—the Narrator, Serial, Parallel, Printer, and Clipboard devices—are disk-resident for all three machines, simply because the Write Control Store ROM (256K) was fully consumed by the other seven devices.

Programming Procedures

The programming procedures for accessing the device internal routines of all 12 Amiga devices are basically the same: a task opens a device unit with an Exec library OpenDevice

call and closes it with an Exec library CloseDevice call. The first time a device unit is operjed, the system automatically allocates and initializes a Device structure to manage the device and a Unit structure to represent the device-unit message port. With shared access modfe devices, the same Device and Unit structures are shared by any tasks that have the devifce's unit open. The Device structure Ub_OpenCnt parameter and the Unit structure unit! OpenCnt parameter are initialized to 1 when the device unit is opened. These parameters are incremented or decremented by 1 each time a task opens or closes the unitL respectively.

A task describes its data needs by using an I/O request structure, which is an extended Exec Message structure containing information on what data the task needs, how fast it needs the data, where to place the data in memory, and how the task can interpret error conjlitions. Generally speaking, when dispatched, the request automatically goes around a loop. From the task's memory space, it goes into the device-unit request queue, into the meijnory space of the device internal routines, back to the task reply-port queue, and finally returns to the task's memory space. A task is not allowed to access the I/O request structure or ks data until this sequence is complete.

However, the system provides command-dispatching mechanisms that can avoid queuing at one or both ends of the transaction; if these mechanisms are used, the task receives thd data sooner than it would with queuing at both ends of the transaction.

A task interacts with a device's internal routines by sending commands to those routines. Commands specify the type of operation required by the task. They are dispatched with the Exec library DoIO and SendIO functions and the individual device library BeginIO functions; these are described in Chapter 2. In the most general sense, a task is either reading data from a device or writing data to a device; almost all other device operations lead up to these operations. Figure I.I illustrates general read-write operations.

If a task uses a buffer in its own memory space to define data and then transmits the data to a device, the system is said to be executing a *write operation.* Here, the data originates in the task-defined buffer in the task's memory space, passes through the device's memory space, and then is sent out to external hardware, where it is permanently stored. Individual devices that have a write capability (the Audio, Serial, Parallel, Narrator, Pointer, Clipboard, and TrackDisk devices) have at least one write command, which is indicated by the word "WRITE" somewhere in the command name.

If a task requests data from external hardware and uses a task-defined buffer to receive the data, the system is said to be executing a *read operation.* The data usually passes through the device's memory space, passes into the task's memory space, and is placed in a task-defined buffer for further access by the task. Individual devices that have a! read capability (the Serial, Parallel, Narrator, Keyboard, Gameport, Clipboard, and TrackDisk devices) have at least one read command, which is indicated by the word "READ" somewhere in the command name.

Devices that allow read operations often have a device internal buffer in RAM that is allocated and managed automatically by the device internal routines. The device is said to "own" that RAM space, even though its internal routines may be in ROM. Devices that allow write operations have a device internal buffer that is allocated and managed automatically by the device internal routines. This buffer also is situated in RAM, even though the device internal routines are sometimes entirely in ROM. The programmer deals only

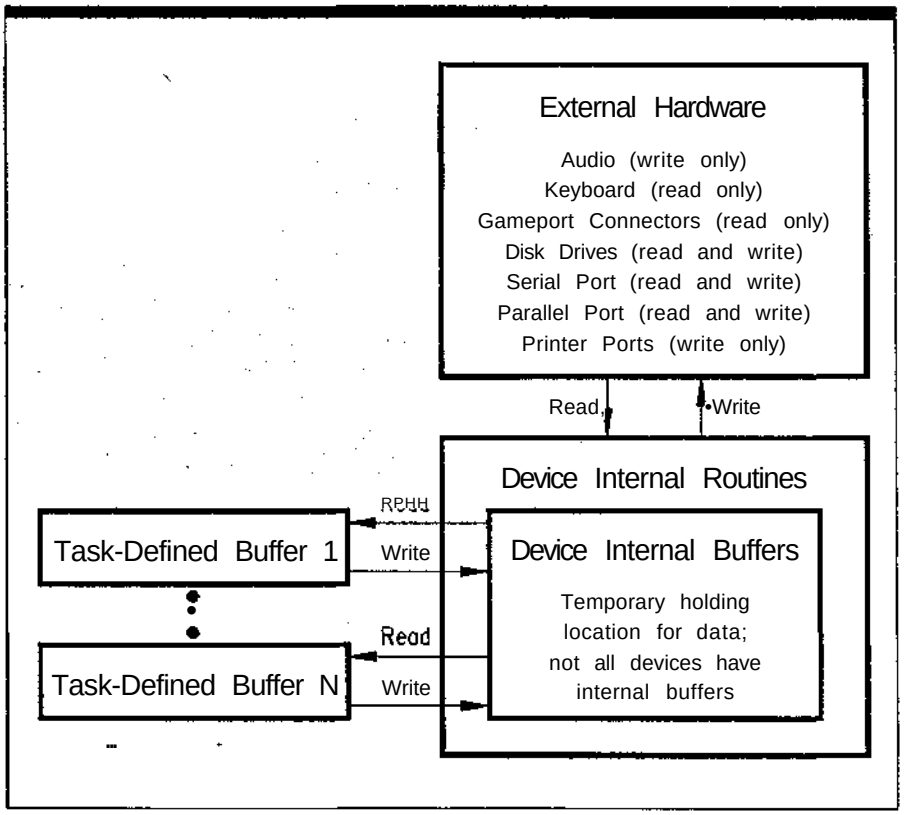


Figure 1.1: *General Read-Write Operations*

with task-defined buffers. Except for task-defined clearing and updating operations, the device internal buffers are managed by the device internal routines.

Most devices return error values when something goes wrong during I/O request processing. Errors are returned in two ways. The OpenDevice, DoIO, and WaitIO Exec library functions return an error value as part of their function call syntax (see Volume I). In addition, detailed, device-specific error values are returned in replied I/O request structures. Generally speaking, a task need only test for a returned nonzero value in the function call to determine if the function executed successfully. If the value is not 0, an error condition occurred during I/O processing, and the task should deallocate all memory and notify the user or take some other appropriate action.

A more detailed level of error checking is also available. Any device processing errors will cause the return of a nonzero value in the I/O request structure io_Error parameter. For example, if an OpenDevice call fails, its I/O request structure io_Error parameter is set to IOERR_OPENFAIL to indicate that the device could not be opened. In most cases, the io_Error value provides the task with all the information it needs in order to determine what went wrong. The task can compare the io_Error value to the preset values in the INCLUDE files and thereby determine the reason the I/O request was

unsuccessful. However, since the programmer must define the detailed comparison and take proper corrective action, in most cases this step is not necessary.

Task-Device Sharing

Because the Amiga is a multitasking system, tasks can often share device units. The following guidelines were built into the system and should be observed for any devices you may want to add:

- If a device sends data to or receives data from external hardware, the system generally provides the device with a mechanism that allows tasks to open it in exclusive access mode. The Serial and Parallel devices both work with hardware, so the system provides each with an explicit flag parameter bit that the task can set before that device is opened; another task cannot open it until the first task closes it. Because the Printer device opens the Serial and Parallel devices indirectly and sends data to a printer connected to the serial or parallel port, it too operates in exclusive access mode, as does each Gameport device unit. (Devices often have more than one unit.) The TrackDisk device is also an exclusive access mode device by default.

- If a device never interacts directly with external hardware except to read data, its units can always be shared among tasks. The Input, Console, Keyboard, Timer, and Clipboard devices operate in this way. The Narrator device sends its data to the Audio device, so it too operates in shared access mode. Although the Audio device sends data directly to external hardware, it provides a complex set of rules that allow multiple tasks to share it also.

The Amiga Programming Environment

Figure 1.2 illustrates the Amiga Release 1.2 programming environment. It shows three disks: the Amiga 1000 Kickstart disk and two C language disks that allow program development. The arrangement of the last two disks applies equally well to all three Amiga machines. This discussion is presented in terms of the Lattice C language compiler, but it is valid for other C language compilers as well—only the names of the compile-and-link programs will vary.

Certain files and information are required to support C language programming, and they must be placed on disk at directory locations that the programming system will recognize. You should start with the Workbench disk in the internal drive and the C language compiler disk in the external drive. Tailor the contents of these disks whenever possible to save disk space; the following sections present useful advice on what is essential and what can be trimmed.

In addition, if you have enough extra memory to create a RAM disk that is large enough to hold most of the files needed for C language programming, you should create a startup-sequence script file to create the RAM disk and the appropriate directories on it, as well as transferring all required programming files to it.

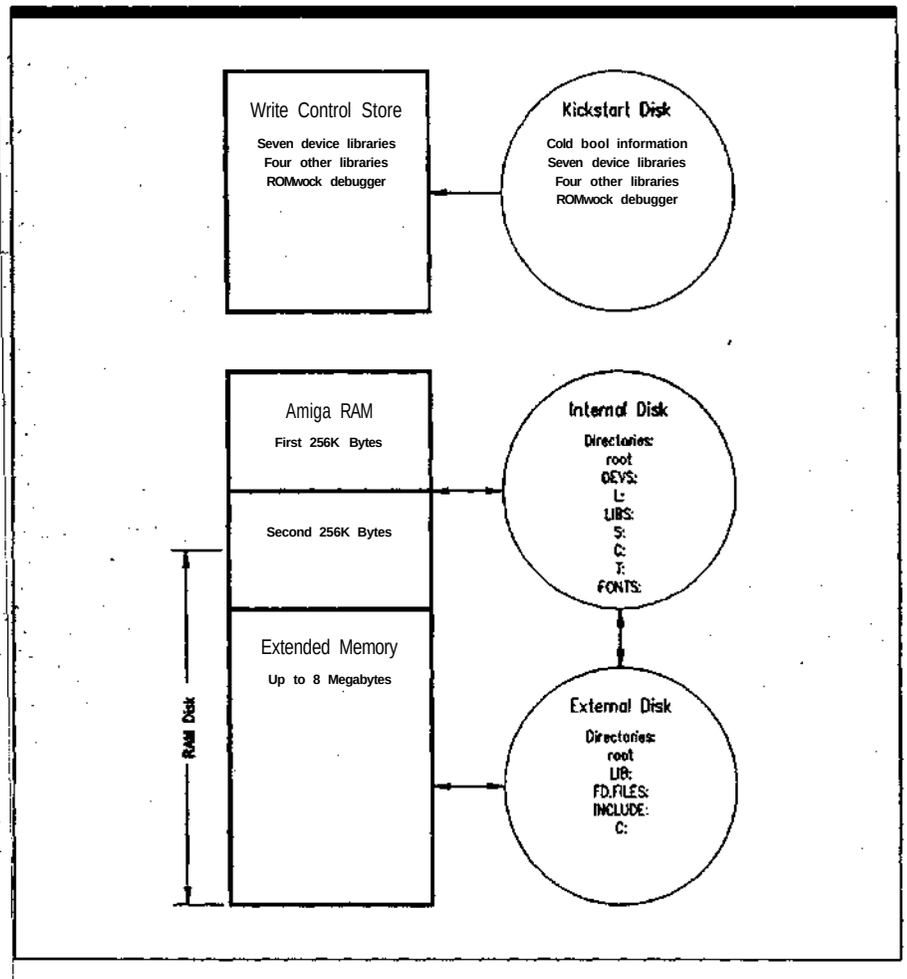


Figure 1.2: *Amiga Programming Environment*

Total Amiga RAM consists of at least 512K of internal RAM and up to 8 megabytes (>f external RAM. The first 512K of internal RAM can be used as chip memory (MEMF-LCHIE, see Volume I), and the external RAM can be fast RAM (MEMF_EAST). An efficient programming system should include at least 512K of internal RAM;256K is not enough. Any additional RAM will also be useful.

The Kickstart Disk

The Amiga 1000 is always cold-booted from the Kickstart disk, which contains several boot sectors and other information required to initialize the memory system and hardware properly. The Kickstart disk also contains most of the Amiga device and library internal routines that you will use in your C language programs and the ROMwac debugger. All

of this information is loaded into a 256K portion of ROM; this is a protected area of memory referred to as WCS (Write Control Store) memory.

On the 500 and 2000 machines, the original Kickstart disk information was placed permanently into ROM. Therefore, for these machines, the Kickstart operating system can only be enhanced by using a ROM chip replacement. With the 1000 machine, once the Kickstart disk has loaded its operating-system information automatically, it is removed from the internal disk drive and the first C language programming disk is then inserted.

The Internal Disk

The C language programming disk contained in the internal disk drive is similar to the Commodore-supplied Workbench disk, but it has been stripped of files not needed for C language programming. However, it must have these directories:

- The root directory. This directory contains any source files for which there is enough memory; you can also leave source files in the root directory on the external disk.
- The DEVS: directory. This directory contains five device libraries not found on the Kickstart disk. It also contains the mountlist file, the system-configuration file, and a printer directory for a group of printer drivers. You should eliminate any printer drivers not needed for your programming. The mountlist file should reflect any disk you want mounted into the system.
- The L: directory. This directory contains three programs necessary for correct operation of the Amiga: the Disk-Validator program, which checks disks as they are inserted and removed from a disk drive; the Ram-Handler program, which manages the RAM disk in which your programming-related files are placed; and the Port-Handler program, which manages the serial and parallel ports.
- The LIBS: directory. This directory contains several library files used in programs that call certain built-in functions. The version.library file manages the library system and keeps track of different programming versions of various libraries.
- The L: directory. This directory contains two libraries, icon.library and info.library, which are not required unless you are using the Workbench functions (see Volume I) to manage icons on the Workbench screen.
- The S: directory. This directory should contain all the AmigaDOS script files you need for your programming. In particular, it must contain the startup-sequence script file, which defines all of the predefined startup operations that must take place when the Amiga is first booted. This file can create the C language programming RAM disk and copy programming-related files onto it. The startup-sequence script file is always executed when the stripped Workbench disk is first inserted and after a keyboard reset sequence. Note that some third-party RAM expansion kits automatically retain the contents of the RAM disk after a reset; with one of these kits, the time-consuming reloading of the RAM disk will not occur each time you encounter a crash and need to reset the machine.

- The C: directory. This direaory contains all AmigaDOS operating-system commands, represented as compiled, executable files. These include the DIR command, the Disk-Copy command, the new AddBufFers command, the Execute command, the Run command, and many other commands necessary to manage files in the AmigaDOS programming environment. You should eliminate any files you will not need during programming; in addition, you can rename most of these command files to save typing time. For example, rename Dir to D, Execute to E, Run to R, and so on.

  You should also place your text-editor program file in the C: directory and copy it to the RAM disk C: directory for greater editing speed. You can then call and execute the editor program no matter what your current direaory happens to be. (While you are doing this, rename your editor so that it is fast to type.)

- The T: directory. This directory contains any temporary files created by the system or other executing programs. Most editor programs place backup text files here automatically. A programmer does not generally access the files in this directory, but takes comfort from knowing that certain files—for example, the last version of an edited source file—are always there as backups.

- The FONTS: directory. This directory contains files that support specific fonts. Topaz is always available directly from the system without appearing in the FONTS: directory, so if you only require Topaz, you can erase all files in this directory to free additional disk space.

The External Disk

The external disk contains the following directories:

- The root directory. This directory can contain any source files that you choose to place on the external disk drive. Source files can be on the internal disk, the external disk, or the RAM disk. The only requirement is that the compile-and-link script files refer to them where they are actually located. If your RAM disk is large enough, you can copy source files to it, together with all other C language-related programming files. If this is done, the C language compile-and-link sequence will be greatly accelerated.

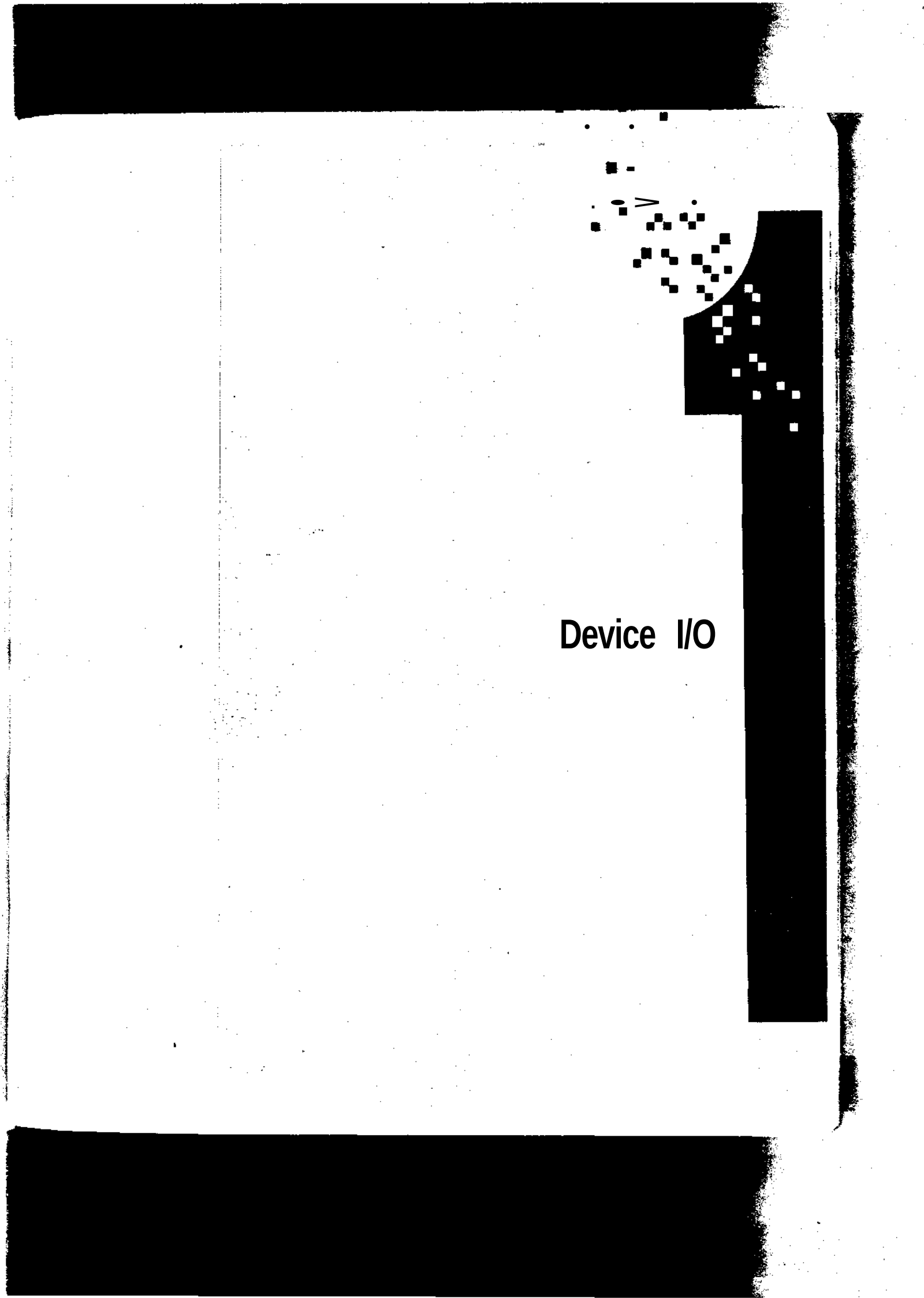
- The C: direaory. This directory contains the C language compiler and linking programs. For the Lattice compiler, these are called LCI, LC2, and Alink. LCI is the first phase of the Lattice C language compiler; it uses your source file as input and produces a quad file as output. The quad file is then used as input to LC2, which produces an object file as output. Alink takes the objea file and produces an executable file as output, together with an error file if any compiler errors occurred during the compile-and-link sequence. The files used in this process and the resulting executable program file will be placed automatically into the disk directory containing the source file; therefore, the programming disk that contains your source files must always have space for these files.

- The LIB: direaory. This directory must contain object files and the information needed to support a compiler's first- and second-pass programs (LCI and LC2).

This includes the Astartup.obj and Lstartup.obj compile-and-link files; the amiga.lib file; the debug.lib file (required only for debugging); and the lc.lib file, which contains compiler-specific functions provided by Lattice. These files are referenced by the compile-and-link script file. The amiga.lib file contains the Exec-support library functions discussed in Chapter 2. In contrast to the other libraries, it is a linked library—a direct reference to it appears in the compile-and-link script file. Therefore, when you define a C language program that references any functions in amiga.lib, you must declare those functions as external (EXTERN) library functions; no OpenLibrary or OpenDevice calls are then needed.

- The FD.FILES: directory. This directory contains descriptor files needed by the compile-and-link programs (LCI, LC2, and Alink) to properly determine function-vector offsets.
- The INCLUDE: directory. This directory contains all the built-in INCLUDE files you will need for C language programming. The INCLUDE files contain structure definitions, flag parameter bit names, other bit definitions, and all other interfacing constants that the C language compiler needs in order to compile and link your program.

The details of the compile-and-link process are described more fully on the disk offered in the back of Volume I.



Device I/O

Introduction

This chapter discusses the general aspects of device I/O and task-device interactions. It presents important cuncepts about tasks and devices. All the functions and standard device commands for the 12 Amiga devices are presented in this chapter also, as well as the appropriate structures and the information in the device-related INCLUDE files. Many of the ideas presented here are extensions of similar ideas in Chapter 1 of Volume I.

When you understand the concepts in this chapter, you will be well on your way to understanding the operation and programming of Amiga devices. You will be able to use the predefined devices efficiently and to add and use your own devices in the Amiga system.

Task-Device Interactions

Figure 1.1 depicts the main interactions between a task and a device. There are several keys to understanding this figure. The first key is understanding how every function works. The functions are discussed in great detail in Volume I. The second key is understanding how each command works. Commands are discussed in detail throughout this volume. The third key is understanding the difference between queued I/O and quick I/O. Figure 1.1 makes this difference obvious. The fourth key is understanding the difference between synchronous I/O and asynchronous I/O, which is described in Volume I and in this volume.

Note that you should study Figure 1.1 beside Figure 1.2 of Volume I. You will then see that task-device interaction is nothing more than a specific instance of task-task interaction, where the routines of the second task are predefined in the system software and arranged into a device library.

The usage and behavior of the MsgPort and Message structures and task signals discussed in Volume I apply equally well here. The most notable exceptions are as follows:

- Device-routine signaling is handled internally and automatically by the device internal routines.
- I/O request replies are handled internally by the device internal routines using the ReplyMsg function.
- The decision to process an I/O request as a quick I/O request is made by the device internal routines. You may request quick I/O, but if the device is not able to process the request as such, it will be treated as a queued I/O request.

If you study Figure 1.1 along with the definitions of the IORequest, IOStdReq, MsgPort, Message, and Unit structures, you will see how each task in the system can communicate with a single device unit.

Describing Figure 1.1 in terms of the IORequest and IOStdReq structures simplifies the discussion. Some devices use these structures directly; however, some use them as substructures in a device-specific I/O request structure. For example, the Serial device uses

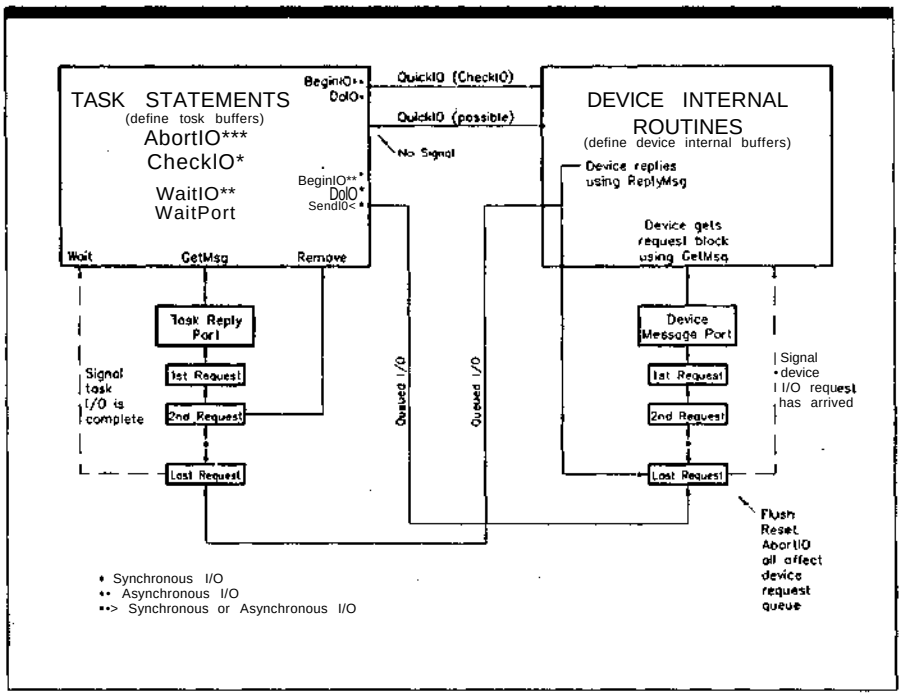


Figure 1.1: ***Task-Device Interaction Using I/O Request Structures***

the IOExtSer structure, and the Parallel device uses the IOExtPar structure, both of which have an IOStdReq substructure as their first entry.

Figure 1.1 shows one task, represented by the large rectangle on the left, and one device unit, represented by the large rectangle on the right. These rectangles represent Amiga C language (or assembly language) programming statements. They do not represent Task structures, even though some of the task statements may include parameter initialization.

The large task rectangle represents all the task statements that define the task, including those dealing specifically with the task-device interaction and the specifying, sending, and processing of I/O requests. In particular, these include the OpenDevice, CIoseDevice, BeginIO, DoIO, SendIO, AbortIO, ChecklO, WaitIO, Wait, Remove, WaitPort, and GetMsg function call statements. These statements also include all the structure parameter definitions of the IORequest or IOStdReq structure required to define each I/O request. One of the most important parameters in these structures is the io_Data parameter, which specifies the task-defined buffers used to pass data from the task to the device and from the device to the task. Chapters 3-14 will discuss how each task defines and manages these buffers.

The device-unit rectangle represents the internal routines of one device unit. Each device unit uses the same set of internal device routines, shared by ail tasks that call on all units of the device. The Device structure is used to manage the library of device internal routines. In addition, the Unit structure is used to manage each unit of the device; it provides a definition of the device I/O request port for that device unit.

Each unit of each device also has a set of internal device buffers used to process the I/O requests coming from each task in the system. These buffers are defined and controlled by the device internal routines; they are not under your programming control. They represent intermediate locations for the data passing back and forth between a task and other areas in the system (in particular, external hardware).

You can think of device internal routines as a set of predefined task routines contained in a predefined library. Recall that a device library is managed by a Device structure, which is equivalent to a Library structure in the Exec software system. These device routines may either be in ROM (ROM-resident device) or brought into RAM from disk (disk-resident device) when the device is opened with the OpenDevice call in the task during the compilation process. Volume I explains how Library and Device structures are defined and managed.

The device-unit I/O request port and the task reply port are represented in Figure 1.1 by smaller rectangles below the two large rectangles. These rectangles represent the list of I/O requests in each message port. A series of queued I/O requests is represented by still smaller rectangles.

The lines in the figure depict the flow of information between a programmer-defined task and the device-unit internal routines. First, consider the line that proceeds from the task rectangle to the bottom of the device request queue. The BeginIO, DoIO, and SendIO functions in the task rectangle send I/O requests from the task to the device-unit request queue.

The line that proceeds from the device rectangle to the bottom of the task reply-port queue indicates I/O requests replied internally by the built-in ReplyMsg function. These were queued I/O requests, and they are being replied to by the device internal routines using the ReplyMsg function internally. The next line represents I/O requests that had the IOF_QUICK flag set. These requests were intended to operate as quick I/O, but the device could not handle them in this way. Instead, the system made these queued I/O requests, and they were also replied to by the ReplyMsg function executing inside the device internal routines.

Device I/O Request Classes

Device I/O requests divide into two classes: queued I/O and quick I/O (most often referred to here and in the Amiga documentation as QuickIO). A third class, for which immediate-mode commands are used, operates automatically. It will be discussed in Chapter 2.

Queued I/O

In queued I/O, each task sends a request to a specific device unit and that request is queued in the device request queue for that device unit. I/O requests are then processed when they reach the top of the device request queue.

The list management is done internally and automatically by the device internal routines when they call the Exec GetMsg function. The routines begin processing the I/O request at the top of the unit queue when the internal GetMsg function returns. Only

when the ReplyMsg function executes in the internal device routines will the requesting task receive data back from the device.

Once the device replies to the I/O request, it is placed in the task reply-port queue. The task can then execute a GetMsg (or Remove) function call to access the replied I/O request.

Queued I/O requests divide further into synchronous I/O requests (sent with the DoIO or BeginIO function) and asynchronous I/O requests (sent with the SendIO or BeginIO function). Note that all types of queued I/O can cause signals to be sent to the requesting task when the device I/O is completed. The signal-passing mechanism is managed by the task-defined MsgPort structure, which is a substructure inside the IORequest or IOStdReq structure.

The task only needs to call the GetMsg function if it sends an asynchronous I/O request with the SendIO or BeginIO function. It must call GetMsg after it has verified that the device has completed the I/O request and the replied 10 request has arrived at the task reply port. The task can use the CheckIO function for this purpose.

In addition, if the task sends an asynchronous I/O request with the BeginIO or SendIO function and calls the WaitIO function to wait for the reply message in the task reply port, WaitIO will also remove the replied 10 request from the task reply-port queue. On the other hand, if the task sends a synchronous I/O request with either the BeginIO. or DoIO functions, these two functions will also perform the job of pulling the replied I/O request from the task reply-port queue.

The Exec Wait function allows any task to wait for I/O request completion signals. In addition, the Wait Port function allows each task to wait for the reply of an I/O request. See Volume I for the distinctions between these two methods.

QuickfO

The second major class of I/O is quick I/O, which will be referred to throughout this volume as QuicklO. With successful QuicklO, no device queuing is done for the I/O request. Instead, the IOF_QUICK flag parameter (io_Flags) of the IORequest structure tells the device that the requesting task wants the I/O to be done quickly. The device will perform the I/O immediately if possible and send the result back to the requesting task.

If the I/O request is successful, it is not placed in the task reply-port queue, nor is the task signaled of the completion of I/O; after all, the required data comes back immediately and the task does not need a signal. If, on the other hand, the device cannot perform the I/O as QuicklO, the request will be queued in the device-unit I/O request queue and will be treated like any other queued request. Each device decides whether QuicklO is possible based on current conditions in the system. If the device is not currently busy, QuicklO usually can occur as requested.

Both the task and the device generally have only one I/O request queue, although the task can have any number of reply-port queues. The parameters in the IORequest, IOStdReq, MsgPort, Message, and Unit structures determine where each I/O request is sent and then replied.

The device-unit request queue includes all I/O requests coming from this particular task and from any other task in the system that sends requests to that device unit. Each task reply-port queue contains all I/O requests replied by this device and any other

devices in the system that reply to this port. The reply port is always specified by the mn_ReplyPort parameter of the Message substructure in the IORequest or IOStdReq structure.

The device-unit queue maintains a list of all I/O requests coming from this task and any other tasks in the system that communicate with this particular unit of the device. Each time an I/O request arrives in the device-unit queue, the device internal routines are automatically signaled of its arrival. The device-routine signaling mechanism is handled internally and automatically by the device internal routines.

The device-unit queue is managed by a set of two standard device commands (CMD_FLUSH and CMD_RESET) and one Exec function (AbortIO). Each device chapter discusses these commands and functions.

Interactions with Multiple Ports and Units

This section explains, with the aid of two diagrams, how multiple reply ports and multiple device units are handled in the Amiga system.

Multiple Reply Ports

Figure 1.2 shows a task working with one unit of a device but using a number of reply ports to handle different types of data coming back from the device. This could be a useful configuration if, for example, you were working with the Serial device and wanted to place data in different categories.

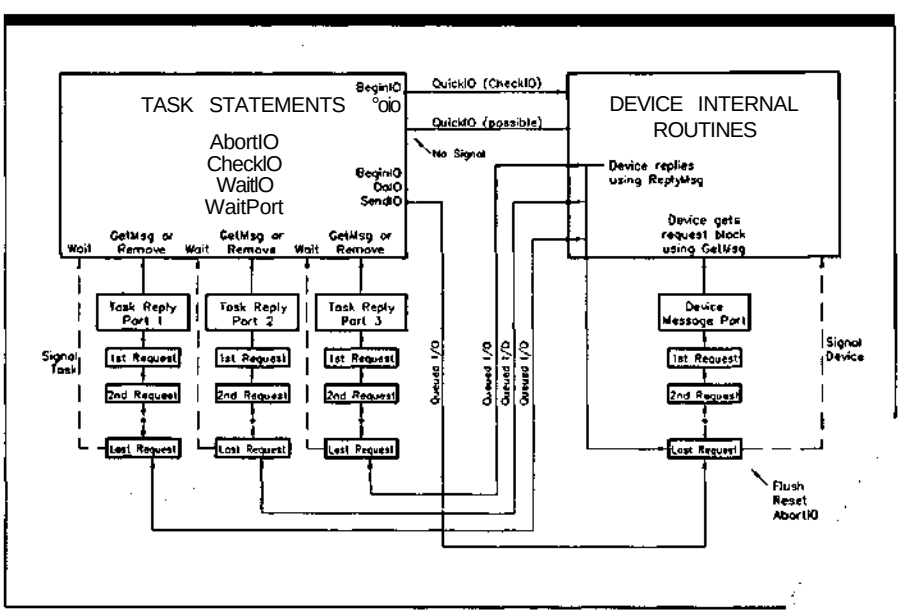


Figure 1.2: *Task-Device Interaction with Three Reply Ports*

Again the two large rectangles represent the task routines and the device internal routines. Below the task rectangle, however, are three task reply ports, each with its own set of replied I/O requests. Each of these reply ports will receive a different category of replied I/O requests when the device is finished processing the I/O requests in that particular category.

You have complete control over which reply goes to which reply port. Each task can define this situation by properly specifying the mn_ReplyPort parameter in the Message substructure of the IORequest or IOStdReq structure when that task initializes these structures before sending a command to the device. Each task can also set up three different task-defined buffers by using the io_Data pointer parameter in the IORequest or IOStdReq structure.

Note that other than the difference involving the mn_ReplyPort parameter, the task-device interaction defined by Figure 1.2 follows the same logic as that in Figure 1.1.

Multiple Units

Figure 1.3 shows one task working with multiple units of a single device. Each device unit could be replying to one or more task reply ports. This is a useful configuration if, for example, you are working with all four units of the TrackDisk device and you want to place data from each disk drive in different task reply ports and associated data buffers.

Queue Behavior

This section explains, with the use of two diagrams, how task reply-port queues and device I/O request queues are used in the Amiga system.

Task Reply-Port Queue Behavior

Figure 1.4 illustrates how a task reply port behaves as one or more devices send their request replies (IORequest and IOStdReq structures) to it.

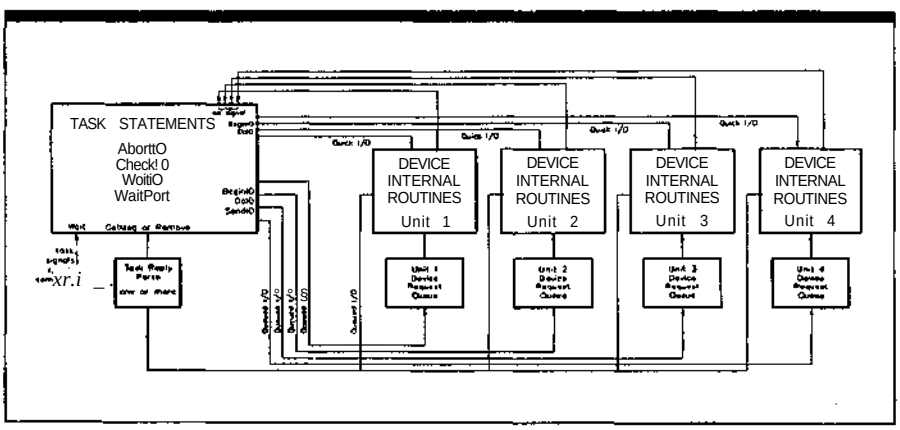


Figure 1.3: *Task-Device Interaction with Multiple Units and Reply Ports*

The task's reply port shows the reply-port queue at one particular point in time. Initially, it contains five I/O requests. Each of these could have come from any of the devices in the system whose IORequest or IOStdReq structure (or other device-specific I/O-related structure) had an mn_ReplyPort parameter that specified a pointer to this task reply port.

The state of the reply-port queue after the GetMsg function finishes execution and returns is shown next. If the original I/O request was an asynchronous I/O request sent with either the SendIO or BeginIO function, then the GetMsg function can be used to remove it from the task reply port. If it was an asynchronous I/O request sent with the SendIO function and the sending task called the WaitIO function, then the WaitIO function will wait for its return and also remove it from the task reply-port queue. If the original I/O request was a synchronous I/O request sent with the DoIO function, then the DoIO function will automatically remove it from the task reply port when a reply is sent by the device.

The next diagram in Figure 1.4 shows the condition of the task reply-port queue after the Remove function has removed IORequest4 from the task reply-port queue. You normally only use this function if your task has one reply port. First the task would call the CheckIO function to see if the I/O request was present in the task reply-port queue. Once the task verifies that the reply message is present in the queue and it gets a pointer to the IORequest structure, the task can call the Remove function to remove it from the queue. The CheckIO-Remove combination is equivalent to the operation of the GetMsg function.

The last diagram in the figure shows the state of the task reply-port queue after the device completes and replies another I/O request. IORequest6 has been added at the bottom of the task reply-port queue. It could have come from any of the devices in the system that processed an I/O request whose reply was addressed to this task reply port.

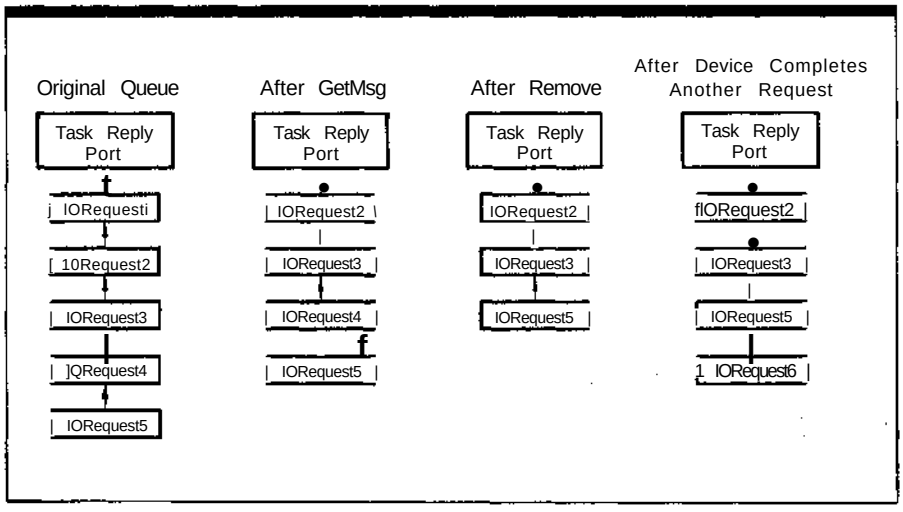


Figure 1,4: *Behavior of a Task's I/O Reply-Port Queue*

Device Message-Port Queue Behavior

Figure 1.5 illustrates how a device unit's message port behaves as one or more tasks send their I/O requests to it.

First you see the device's message port showing the I/O request queue at one particular point in time. Initially, there are five I/O requests queued; each could have come from any of the tasks in the system whose IORequest or IOStdReq structures (or device-specific I/O-related structures) were sent to that device.

Next you see the state of the queue after the device processes the first request. IORequest 1 has been removed from the queue—the device internally uses the equivalent of the GetMsg function to remove it—and the device is in the process of satisfying the request.

The next diagram shows the condition of the queue after the BeginIO, DoIO, or SendIO function has queued another I/O request (IORequest6) in the device request queue. This request could have come from any task in the system that was communicating with the device unit.

The next diagram shows the state of the queue after a task calls the AbortIO function to remove a pending I/O request that is no longer needed. In this case, IORequest3 was removed. The last diagram shows the state of the queue after the CMD_FLUSH or CMD_RESET command has finished executing. The device request queue has been emptied of all pending I/O requests, and they must be explicitly sent again for the device to process them.

Shared Access versus Exclusive Access

Figure 1.6 illustrates the distinction between shared and exclusive device access when more than one task tries to access a specific unit of a device. Here you see three tasks trying to access the internal routines of a device.

Any device that can be accessed in both shared and exclusive modes has a flag parameter bit that specifies the type of task access requested for that device. For instance, the Serial device has the SERF_SHARED flag parameter bit, which tells the Serial device

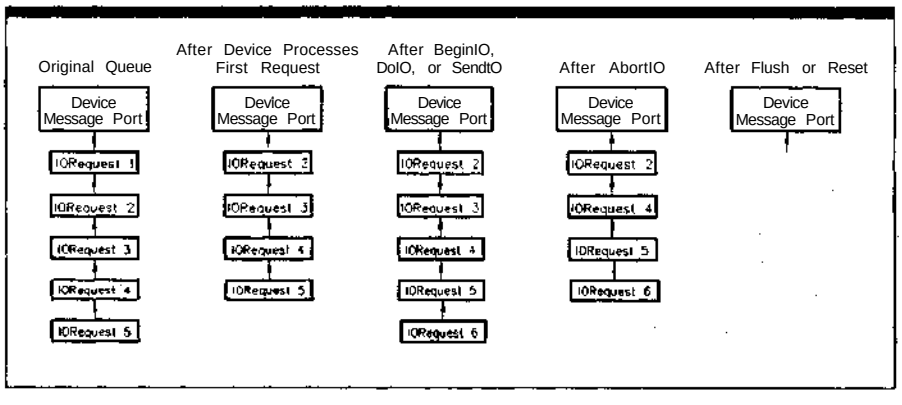


Figure 1.5: *Behavior of a Device's I/O Message-Port Queue*

internal routines that you want to open the device in shared access mode. Note that the default for all devices is not always exclusive access.

The top of Figure 1.6 shows how these three tasks interact with the device internal routines when all three tasks open the device unit with the flag parameter bit for shared access mode specified. Here each of the these tasks sends I/O requests to the device internal routines (using BeginIO, DoIO, or SendIO) after they have opened the device with the OpenDevice function.

Task switching is not prevented while these three tasks send I/O requests and receive replies for the device-generated data. After the first task calls OpenDevice for that unit of the device, another task can also call OpenDevice and request data from that unit. There is no need for Taskl to close the device unit before Task2 and Task3 can open the device and send I/O requests to the same unit.

Exclusive access operates in a different way, as the three diagrams at the bottom of Figure 1.6 show. Here Taskl must finish using the device before Task2 and Task3 can gain access to it. All of the BeginIO, DoIO, and SendIO functions in Taskl must be surrounded by a pair of OpenDevice and CloseDevice function calls before task switching allows another task to access that unit of the device.

This does not mean that task switching is prevented; it only means that if a task switch occurs before Taskl has closed the device unit, any attempt by Task2 or Task3 to open that same device unit will result in a failure to open. The task will return IOERR_ OPENFAIL at least until the first task regains the CPU and closes the device unit. In Figure 1.6, Task2 will not be able to open the device until Taskl regains the CPU, executes a CloseDevice

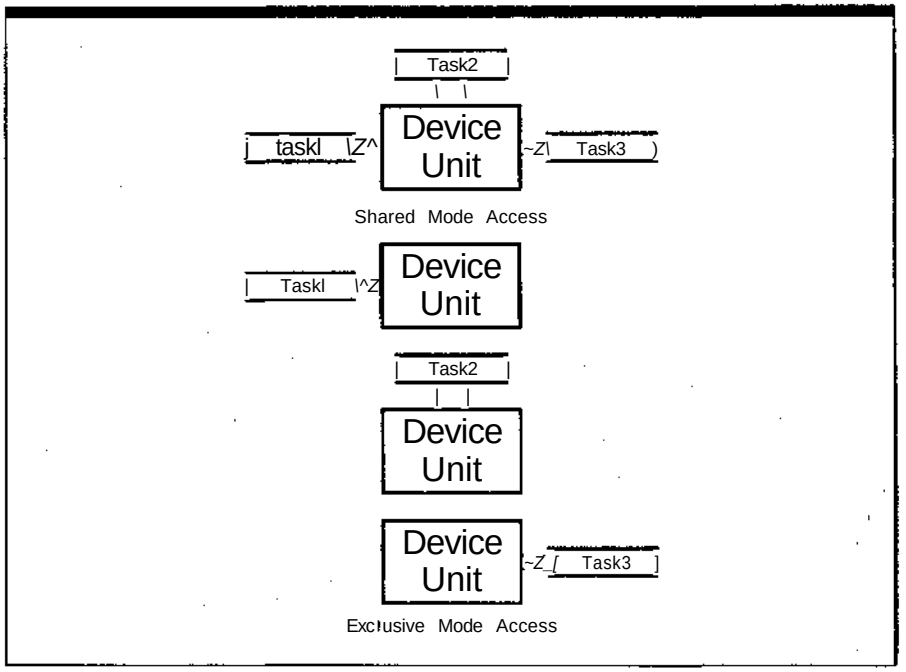


Figure 1.6: *Difference between Shared and Exclusive Access*

call, and closes that unit of the device. Once Taskl closes the device, the OpenDevice call in Task2 will succeed when Task2 once again regains the CPU.

These ideas are important in developing your programming logic. You must decide on a task-device unit opening and closing sequence for ail your tasks. First you must decide the access mode for each device unit you intend to open in each task; then you must decide when you want to open and close each device unit.

Both the Device and the Unit structures have a structure parameter that keeps track of the number of tasks that have opened a device unit and subsequently closed it. In the Device structure this parameter is called lib_OpenCnt, and in the Unit structure, it is called unit_OpenCnt. The system works with these parameters, together with the type of access you specify in your programs, to determine what action to take when a task tries to open a device unit.

One way to simplify such decisions is to use all available units of a device to avoid task collisions for the same device unit. For example, for the TrackDisk device, if you have two or more disk drives you can establish a strategy for using those drives in the most efficient manner. The Audio device discussed in Chapter 3 provides a good illustration of using four units simultaneously to produce complex stereo sounds.

Multitasking and I/O Request Processing

Figure 1.7 illustrates the details of multitasking when a series of tasks sends a series of I/O requests to a specific device unit. This figure shows the difference between device processing for asynchronous and synchronous I/O requests.

Three tasks (Taskl, Task2, and Task3) are communicating with the same device unit. A typical example would be three tasks communicating with the Serial device, each trying to get its own category of data from the Amiga serial port. Taskl needs to send three I/O requests to the device unit: IORequestll, IORequestl2, and IORequestl3, shorthand notations for the complete IORequest or IOStdReq (or, for the Serial device, IOExtSer) structure used to define the I/O request. Task2 needs to send IORequest21j IORequest22, and IORequest23 to the device unit. Task3 needs to send IORequest31, IORequest32, and IORequest33.

In this example, Taskl has the highest task priority (ln_Pri = 60), Task2 the next-highest (ln_Pri = 55), and Task3 the lowest (ln_Pri = 50). Each of these tasks has opened the device unit with an OpenDevice function call, and each task opened the device unit in shared access mode. These arrangements allow for task switching and device sharing. Finally, the device request queue is presently occupied by a number of queued I/O requests previously placed there by other tasks in the system.

The three tasks go through the following series of steps:

1. Taskl issues a DoIO call to send IORequestll. Recall that DoIO initiates a synchronous I/O request. Because the device-unit request queue is not empty in this example, the device unit will not be able to immediately service this request; it will be queued behind other already present I/O requests. Because IORequestll cannot be processed immediately and DoIO cannot return in Taskl, Taskl will be blocked. The next-higher priority task will take over; by assumption, this is Task2.

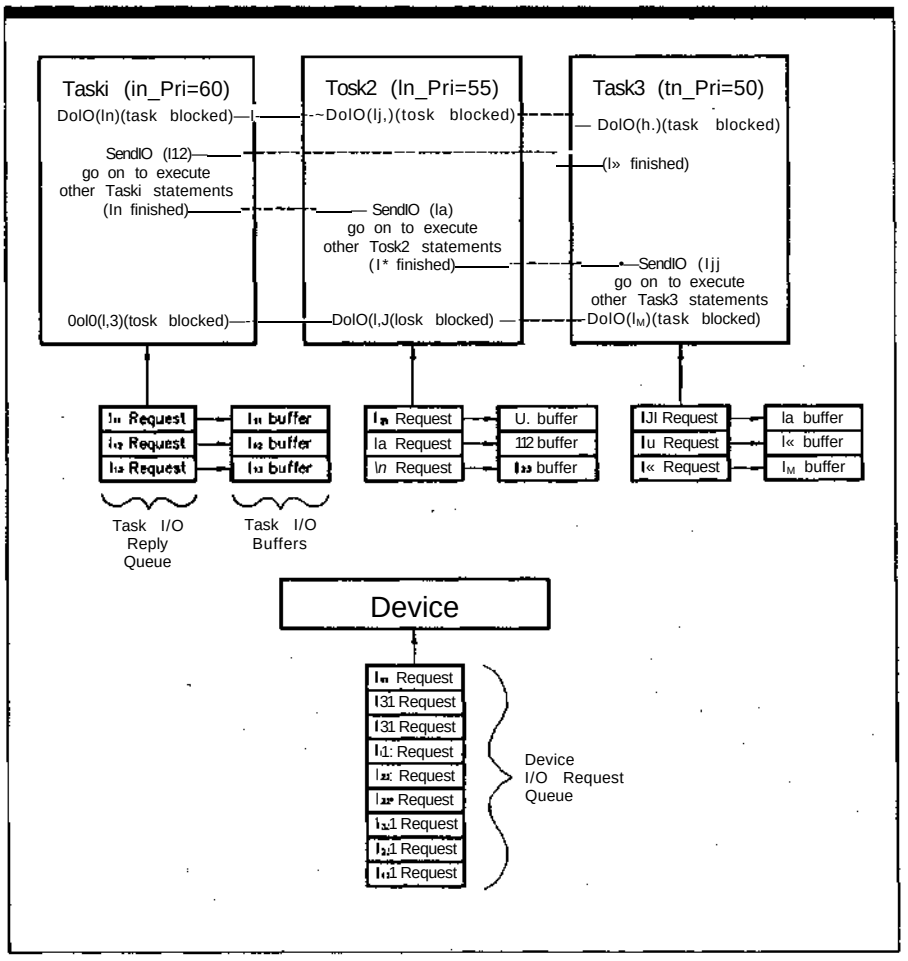


Figure 1.7: *Multitasking and I/O Request Processing*

2. Task2 gets control of the machine and sends the IORequest21 synchronous I/O request to the same device unit using a DoIO function call in Task2. IORequest21 is queued behind IORequestll and other requests already in the queue. Task2 will now be blocked.

3. Task3 gets control of the machine and sends the IORequest31 synchronous request to the device unit, using a DoIO call. This request will now be queued behind IORequestll, IORequest21j and other I/O requests already in the queue. Task3 will now be blocked.

Now assume that the device internal routines just finished with IORequestll. (This is an assumption about the sequence and timing of events in the system, not something you can directly control. Note that the previously queued requests must also be processed

before IORequestll is processed.) The previously blocked Taskl can then get control of the CPU; Taskl has been waiting on IORequestll, and the device internal routines have signaled that IORequestll is completed. Taskl receives the IORequestll request in its task reply-port queue and acts on the arrival signal. This gives CPU control to Taskl, indicated by the dotted line between the Taskl and Task3 rectangles. The device internal routines will signal Taskl (using the equivalent of the PutMsg function's signal mechanism) and Taskl can now go on to execute other task statements.

Now assume that the next task statement in Taskl is a SendIO function call. This call sends IORequestl2 (asynchronous) to the queue, behind IORequest21, IORequest31, and other previously queued I/O requests. (Note that IORequestll is no longer in the device-unit request queue).

Because SendIO sends an asynchronous I/O request, Taskl can now go on to execute other task statements. Here, however, the device unit has just finished processing IORequest21. (This is again an assumption about the specific sequence and timing of events over which you have no direct control.)

Now that IORequest21 is completed, Task2, the next-highest priority task, can once again gain control of the CPU, as shown by the dotted line between the Taskl and Task2 rectangles. Assume that the next executable task statement in Task2 is a SendIO function call; the process continues from here in the same way.

If you study this diagram along with the discussions of DoIO, SendIO, and other I/O functions in Volume I, you can see how both asynchronous I/O requests and synchronous I/O requests are handled in the Amiga system. These considerations have a bearing on the design of your programs and the design of all of the tasks that make up those programs.

Amiga Devices

Figure 1.8 shows the relationship between a task and the 12 predefined Amiga devices in the Amiga system. Keep in mind that the task depicted by the large rectangle represents any task in the system, either a programmer-defined task or a system task.

The large rectangle represents all the task statements within that task, including those that communicate directly with the devices. These device statements include all OpenDevice, CloseDevice, BeginIO, DoIO, SendIO, AbortIO, WaitIO, ChecklO, WaitPort, GetMsg, Remove, and Wait function calls that are directed at a device-unit I/O request queue or a task I/O request reply-port queue.

Twelve smaller rectangles in the figure represent specific units of a device. Remember that each device could be shown as many rectangles, each representing a different unit of the device, with as many rectangles as that device has allowable units. For example, the rectangle for the TrackDisk device could be expanded to four rectangles, one for each of its four possible units. In addition, the Translator library, which is not a device, is also shown as a rectangle; it is included because it works directly with the Audio and Narrator devices.

The most important things to note about Figure 1.8 are as follows:

- With enough memory, each task can open up to 12 predefined devices for simultaneous access as those tasks each switch in and out of execution. A task may be able

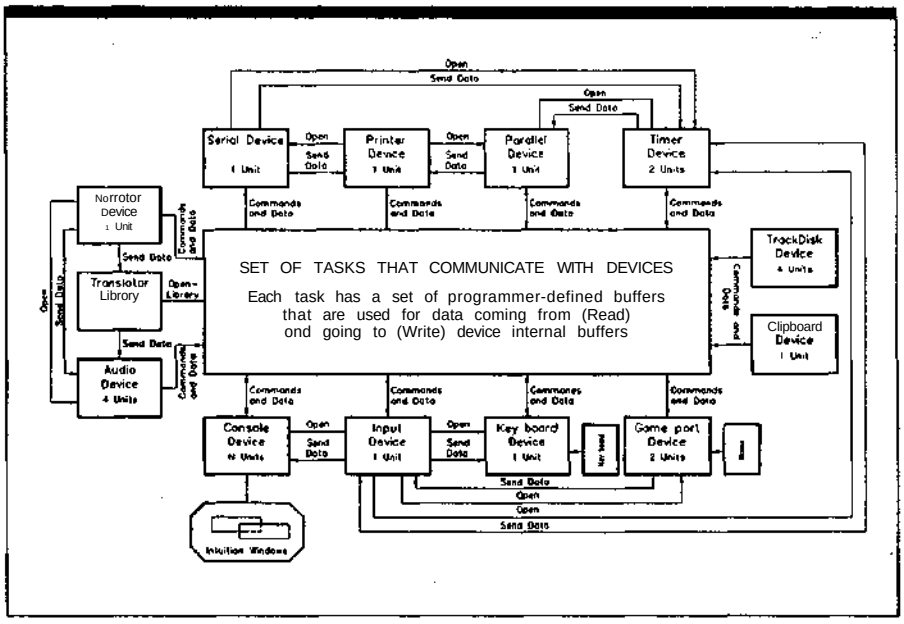


Figure 1.8: *Task-Device Relationships for AH Amiga Devices*

to open more than one unit of each of the 12 devices; in fact, it may be able to open all units of all devices. The main limitation is memory. If all units of all devices are open and the system is very active, there will be a lot of queued I/O requests, which use a lot of RAM. The number of units allowed for each device is indicated in the small rectangle representing that device.

- The double-sided arrows from the large task rectangle to the small device-unit rectangles represent task-device interactions—the transfer of all commands and data between the task and the device internal routines brought about by the functions executing within the task. In particular, these arrows represent the OpenDevice, CloseDevice, BeginIO, DoIO, and SendIO function calls.

- The arrows labeled Open and Send Data depict the internal operations and effects of one device on another. For example, the Console device shows an Open arrow running to the Input device. This means that the Console device will automatically open the Input device when any task issues an OpenDevice call to open the Console device. The arrows labeled Send Data each have a similar meaning; when the Console device indirectly opens the Input device, the Input device can send data to the Console device. This device-to-device data transfer is handled automatically by the device internal routines.

Study all the relationships depicted in Figure 1.8. These interactions will be discussed in later chapters.

Standard Device Commands

Table 1.1 summarizes the standard commands for each device in the system. The Amiga provides a maximum of nine standard commands for each device. Note that the number of standard commands actually implemented varies from device to device.

These commands were briefly discussed in Volume I. The main characteristics of each are again presented here:

- CMD_CLEAR clears al! internal device buffers. Recall that each device has a set of internal buffers that it uses to manage data once control is inside the device internal routines. The CMD_CLEAR command tells the system to zero all bytes in each of the device-unit internal buffers. CMD_CLEAR has no effect on the task-defined buffers.

- CMD_FLUSH tells the system to abort all pending I/O requests in the device-unit request queue. Once these requests are flushed, a task will have to initialize the I/O request structures if it needs to send those requests again.

- CMD_IN\^\LID tells the system that the task is sending an invalid command. See Chapter 11.

- CMD_READ tells the system to read a number of data bytes from the device internal buffers into one or more of the task-defined buffers. The number of bytes to read is usually specified by the IOStdReq structure io_Length parameter; the system usually places the number of bytes actually read into the IOStdReq structure io_Actual parameter. There are exceptions to these rules; the details of using this command vary from device to device.

- CMD_RESET tells the system to reset the device unit. It completely reinitializes the device internal routines, returning them to their default configuration. CMD_RESET also aborts all queued and currently active I/O requests, cleans up any data structures used by the device internal routines, and resets any related hardware registers in the system.

- CMD_START tells the system to restart execution of a device command that was previously stopped with the CMD_STOP command. The restarted command then resumes where it stopped. In some cases, however, a command cannot restart at the precise data byte at which it was stopped; the system then chooses another point at which to restart the command.

- CMD_STOP tells the system to immediately stop the data processing currently being done by the device unit. It will stop the processing at the first opportunity. All I/O requests continue to queue, but the device unit stops processing them. The device request queue can grow quickly if a lot of task and system activity occurs while the device is stopped; if this happens, a great deal of memory may be used by the queued I/O request structures. The command is useful for devices that require Amiga user intervention (printers, plotters, and data networks, for example).

Table
Sta
Comn
fot
Amiga I

Device	CLEAR	FLUSH	INVALIO	READ	RESET	START	STOP	UPDATE	WRITE
Audio	✓	✓	1	✓	✓	✓	✓	✓	✓
Disk Card	1	✓	1	✓	✓	1	1	✓	✓
Console	✓	—	—	✓	✓	—	✓	✓	✓
Gameport	✓	—	1	✓	✓	1	1	—	1
Input	—	—	1	✓	✓	✓	1	—	1
Keyboard	✓	✓	1	✓	✓	1	1	—	1
Narrator	1	✓	1	✓	✓	✓	✓	—	✓
Parallel	✓	✓	1	✓	✓	✓	✓	—	✓
Printer	—	✓	✓	1	✓	✓	✓	—	✓
Serial	✓	✓	1	✓	✓	✓	✓	—	✓
Timer	1	—	1	1	1	1	1	—	1
TrackDisk	1	—	1	✓	✓	1	1	✓	✓

Table 1.1: *Standard Commands for Each Amiga Device*

- CMD_UPDATE tells the system to write all device internal buffers out to the physical device unit. The information in these buffers usually originates in the task-defined buffers; the device internal buffers represent a holding location for the task information. The device performs this operation automatically as part of its normal operations; however, this command can also be used to cause an explicit update under the control of a programmer task. It is useful for devices that maintain internal data buffers (caches) such as the floppy-disk and hard-disk drives.

- CMD_WRITE tells the system to write a number of data bytes from a task-defined buffer into one or more of the device internal buffers and then perhaps onto external hardware (for example, a disk). The number of bytes is usually specified by the IOStdReq structure io_Length parameter; the system places the number of bytes actually written into the IOStdReq structure io_Actual parameter. Once again, the details of this command vary from device to device.

Of the 12 Amiga devices, four are disk-resident (the Narrator, Parallel, Printer, and Serial devices), and eight are ROM-resident. In addition to the standard device commands shown in Table 1.1, most devices are programmed with a number of device-specific commands.

Device Functions

The Amiga provides eight standard Exec functions for use with each device. Some devices use several other Exec functions.

All of the devices have an explicit OpenDevice function call. In addition, the Keyboard device is always opened automatically by the Input device, which, in turn, may be opened automatically by the Console device, which is opened automatically by the system upon machine startup or reset. Note that the Console device can only be opened if AmigaDOS is active.

The system automatically creates a ROM-based input task when it is started. This task is used by both the Console device and Intuition. Intuition traps some of the input events, including mouse movements and keyboard events, needed for window input processing.

All of the devices have an explicit CloseDevice function call. However, the Console device is closed by the system upon reset or power down; it also takes the other devices down with it. Automatic closing is necessary to recover system resources—in particular, memory.

Once you understand the relationships between the standard functions and the device-specific functions, you can use them to program the Amiga devices. More particulars about each device function are presented in Chapters 3-14.

Structure Linkages for Tasks and Devices

Figure 1.9 depicts the structure linkages for all structures that are directly related to task-device unit management in the Amiga system. Some devices also have device-specific structures that are used to manage that device.

Figure
Stru
Linkage
General
Devic
Proce

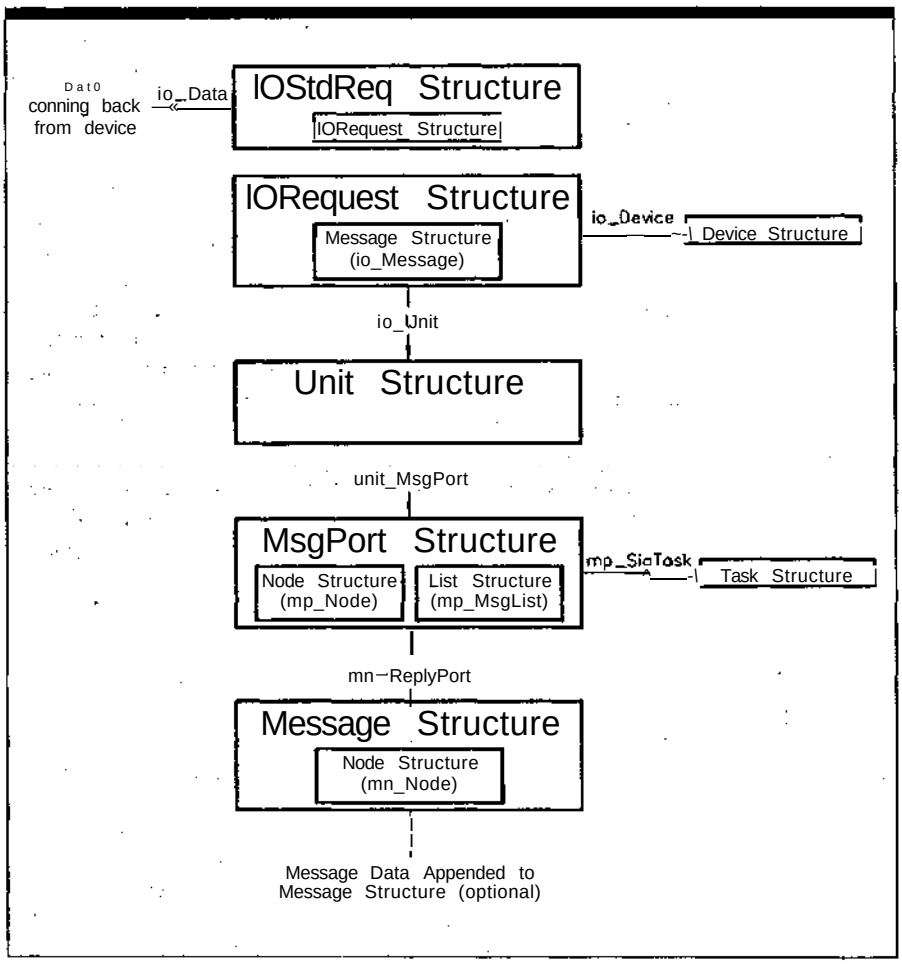


Figure 1.9: *Structure Linkages for General Task-Device I/O Processing*

The IOStdReq structure contains the lORequest structure as a substructure. If the task needs a task-defined data buffer, it must use the IOStdReq structure, which includes the io_Data buffer pointer. For a read operation, the io_Data parameter specifies the task RAM buffer in which the device internal routines will place their data. For a write operation, this parameter defines the task buffer RAM location in which the task should place the data it will send to the device.

The lORequest structure contains the io_Device pointer, which points to a Device structure. A Device structure is identical to a Library structure and is used to help manage the operation of all open device units. This parameter is specified by the system when the OpenDevice function call returns.

The lORequest structure also contains the io_Unit pointer, which point;^©" the—Unit .^ structure used to manage the operation of one device unit. Just like the io£35j^c^£arameter, ~A i. **the** io_Unit parameter is specified by the system when the OperJDevic^Sjhction caUS^turns.M /-

In addition, the IORequest structure contains a Message substructure named io_Message, which is used to define the parameters of the I/O request message. It contains a pointer (mn_ReplyPort) to the task reply (message) port that will receive the I/O request when the device unit sends it back to a task.

It is important to note that there are two MsgPort structures in the I/O system. The first is used for the device I/O request queue, and the second is used for the task reply-port queue. Each MsgPort structure contains a Node substructure (mp_Node) and a List substructure (mp_MsgList). They manage the message list for the two I/O request queues.

The MsgPort structure contains a pointer to a Task structure. For the task-related MsgPort structure, this indicates which task will be signaled when the device internal routines reply one of the I/O requests in the device-unit request queue; they will use the ReplyMsg function to reply and to signal the task of its completion.

The Message structure contains a Node substructure named mn_Node. It is used to place I/O requests on the message list of the device unit's message-port queue or the task reply-port queue.

Any Exec Message structure can always be extended by the addition of optional message data; this data can supplement the normal task-defined buffer data that passes back and forth between the task and the device. You can see that the IORequest and IOStdReq structures (or any device-specific I/O request structures) are nothing more than customized Message structures with appended data.

General I/O Structures in the Amiga System

Dealing with devices in the Amiga system requires the programmer to work with the system's five key structures: IORequest, IOStdReq, MsgPort, Message, and Unit. Each structure has a number of parameters that control the processing of device I/O requests. The required operations include initializing parameters, reading parameters, and writing parameters.

A programmer-defined task must work together with the system routines and the device internal routines to supply and gather the information going to and coming back from devices. For these reasons, the most important features of these structures are now presented.

Refer to Volume I and the appropriate chapters in this volume for more details about these structures and their parameters.

The IORequest Structure

The IORequest structure is defined as follows:

```
struct IORequest {
    struct Message io_Message;
    struct Device *io_Device;
    struct Unit *io_Unit;
    UWORD io_Command;
    UBYTE io_Flags;
    BYTE io_Error;
};
```

These are the parameters in the IORequest structure:

- io_Message. This parameter is a Message substructure containing message information associated with the IORequest structure. The Message structure is used by the device to return your I/O request upon completion. It is also used by devices internally for I/O request queuing in each unit of the device. The Message structure (in particular, the mn_ReplyPort parameter) must be properly initialized for I/O to work correctly.

- io_Device. This parameter is a pointer to a Device structure for the device associated with this IORequest structure. It is automatically set by the Exec system routines when the device is opened with the OpenDevice function. Remember that a Device structure is formally identical to a Library structure, discussed in detail in Volume I. Of particular importance here, however, is the lib_OpenCnt parameter, which the system automatically maintains as the number of tasks that are currently using the Device structure. This is a device-private parameter; once set by OpenDevice, it should not be changed by the calling task.

- io_Unit. This parameter is a pointer to a Unit structure that represents a particular device unit. It is automatically set by the Exec system routines when the device is opened with the OpenDevice function. Of particular importance is the unit_OpenCnt parameter, which the system automatically maintains as the number of tasks that are currently using this Unit structure. This is a device-private parameter; once set by OpenDevice, it should not be changed by the calling task.

- io_Command. This parameter contains the device command to execute. It may be either a standard device command or a device-specific command.

- io_Flags. This is a set of flag parameters for the IORequest structure. The flag parameters are divided into two fields of four bits each. The lower four bits (bits 0 to 3) are used by the Exec system routines; the upper four bits (bits 4 to 7) are available to each device for its own uses. See below for the definition of io_Flags bit 0.

- io_Error. This parameter is an error number returned to the calling task upon I/O request completion or failure. I/O errors fall into two categories: standard device errors and device-specific errors.

The io_Flags flag parameters in the IORequest and IOStdReq structures are as follows:

- IOF_QUICK. Set this if you want to use QuicklO. Then the device will process the I/O request immediately if possible. If the device cannot handle the request as a QuicklO request, it will be queued just as if it had been sent as a queued I/O request. This is io_Flags parameter bit 0. See the specific chapters for other device-specific values of io_Flags.

The IOStdReq Structure

The IOStdReq structure is defined as follows:

```
struct IOStdReq {
    struct Message io_Message;
    struct Device *io_Device;
    struct Unit *io_Unit;
    UWORD io_Command;
    UBYTE io_Flags;
    BYTE io_Error;
    ULONG io_Actual;
    ULONG io_Length;
    APTR io_Data;
    ULONG io_Offset;
    ULONG io_Reserved1;
    ULONG io_Reserved2;
};
```

The first six parameters in the IOStdReq structure—io_Message, io_Device, io_Unit, io_Command, io_Flags, and io_Error—are the same as for the IORequest structure. The other parameters are as follows:

- io_Actual. This parameter usually represents the actual number of bytes transferred during the requested I/O operation. It is only valid upon I/O completion. Not all devices return this value.

- io_Length. This parameter usually contains the requested number of bytes to transfer; each task must set it prior to sending the I/O request. A value of - 1 can be used to indicate variable-length data transfers terminated by some EOF (end of file) condition. EOF characters, where appropriate, are defined separately for each device. Not all devices require this value.

- io_Data. This parameter is a pointer to the task-defined data buffer for task-device data transfers. This is the data buffer over which your task has complete control.

- io_Offset. This parameter is a byte-offset specification for byte-offset-structured devices, such as the floppy disk controlled by the TrackDisk device. This number must be a multiple of the device block size (for example, 512 bytes for a floppy-disk device).

- io_Reservedl and io_Reserved2. These parameters each contain four bytes reserved for future structure expansion.

The Unit Structure

The Unit structure is defined as follows:

```
struct Unit {
    struct MsgPort *unit_MsgPort;
    UBYTE unit_Flags;
    UBYTE unit_Pad;
    UWORD unit_OpenCnt;
};
```

These are the parameters in the Unit structure:

- unit_MsgPort. This parameter is a pointer to a MsgPort structure that is used to queue all I/O requests coming from all tasks into this device unit. The message port will be shared by more than one task if those tasks open the unit in shared access mode.
- unit_Flags. This parameter contains a set of flag parameters for the device unit. See below for the definition of the unit_Flags parameter.
- unit_Pad. This parameter is a one-byte padding that is used to word-align the parameters in the Unit structure.
- unit_OpenCnt. This parameter contains a count of the number of tasks that opened a unit of the device. It is incremented or decremented each time a task opens or closes the unit. The parameter allows the same device unit to be shared by a number of tasks.

The unit_Flags flag parameters in the Unit structure have the following meanings:

- UNITF_ACTIVE. The device unit associated with this Unit structure is currently active accessing its internal routines to process an I/O request.
- UNITF_INTASK. The device unit associated with this Unit structure is currently associated with a particular task. Therefore, if the unit is opened in exclusive access mode, another task will not be able to open it until the other task closes it.

Both of these flag parameter bits are controlled by the system.

The MsgPort Structure

The MsgPort structure is defined as follows:

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

These are the parameters in the MsgPort structure:

- mp_Node. This parameter is a Node substructure that is used to place this message port on the message-port list of all MsgPort structures in the system. The system automatically maintains a list of message ports. The Node structure contains the ln_Name parameter, which can be set with a simple structure-parameter assignment statement. Once the In_Name parameter is defined, this MsgPort structure can be referenced by name by a series of cooperating tasks, each of which can then add or remove I/O requests from that message port.
- mp_Flags. This parameter contains a set of flag parameters for the MsgPort structure.
- mp_SigBit. This parameter is the signal bit number used to signal a task when a message arrives in the message port. Each message port can have only one signal bit number.
- mp_SigTask. This parameter is a pointer to a Task structure that represents the task to be signaled when a message (I/O request) arrives in the message port. This is usually the task that "owns" the message port.
- mp_MsgList. This parameter is a List substructure that maintains a list of all messages arriving in the message port represented by this MsgPort structure. The Message structure Node substructure is used to place nodes on this list.

The Message Structure

The Message structure is defined as follows:

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

These are the parameters in the Message structure:

- mn_Node. This parameter is a Node substructure that allows all messages arriving at a message port to be placed in a message list.
- mn_ReplyPort. This parameter is a pointer to a MsgPort structure that represents the message port to which the message should be sent once the receiving task has accessed or used its data. For task-device interaction, each task must always initialize this parameter before dispatching a command.
- mn_Length. This parameter contains the number of data bytes in the message. It is usually not used for task-device I/O. The message data itself is always appended to the Message structure.

Device-Related Structures and INCLUDE Files

Table 1.2 presents a summary of the device-related structures and the INCLUDE files defining these structures.

Not all of the Amiga devices have a device-specific I/O request-type structure explicitly assigned to them. In particular, four of the Amiga devices—the Console, Gameport, Input, and Keyboard devices—show no such structure in their INCLUDE files. This does not mean that you cannot send commands directly to them; instead, the command-sending mechanism relies on the IOStdReq structure itself.

The other eight Amiga devices have one or more I/O request structures assigned to them. The Printer device has two I/O request structures assigned to it. IOPrtCmdReq is used for sending most I/O requests to the Printer device; IODRPReq is used for sending dump-raster bitmap-to-printer I/O requests.

The Audio and Timer devices both use the IORequest structure as a substructure in their I/O request structures to send commands and data to their respective device routines. On the other hand, the Clipboard, Narrator, Parallel, Serial, Printer, and TrackDisk devices use the IOStdReq structure.

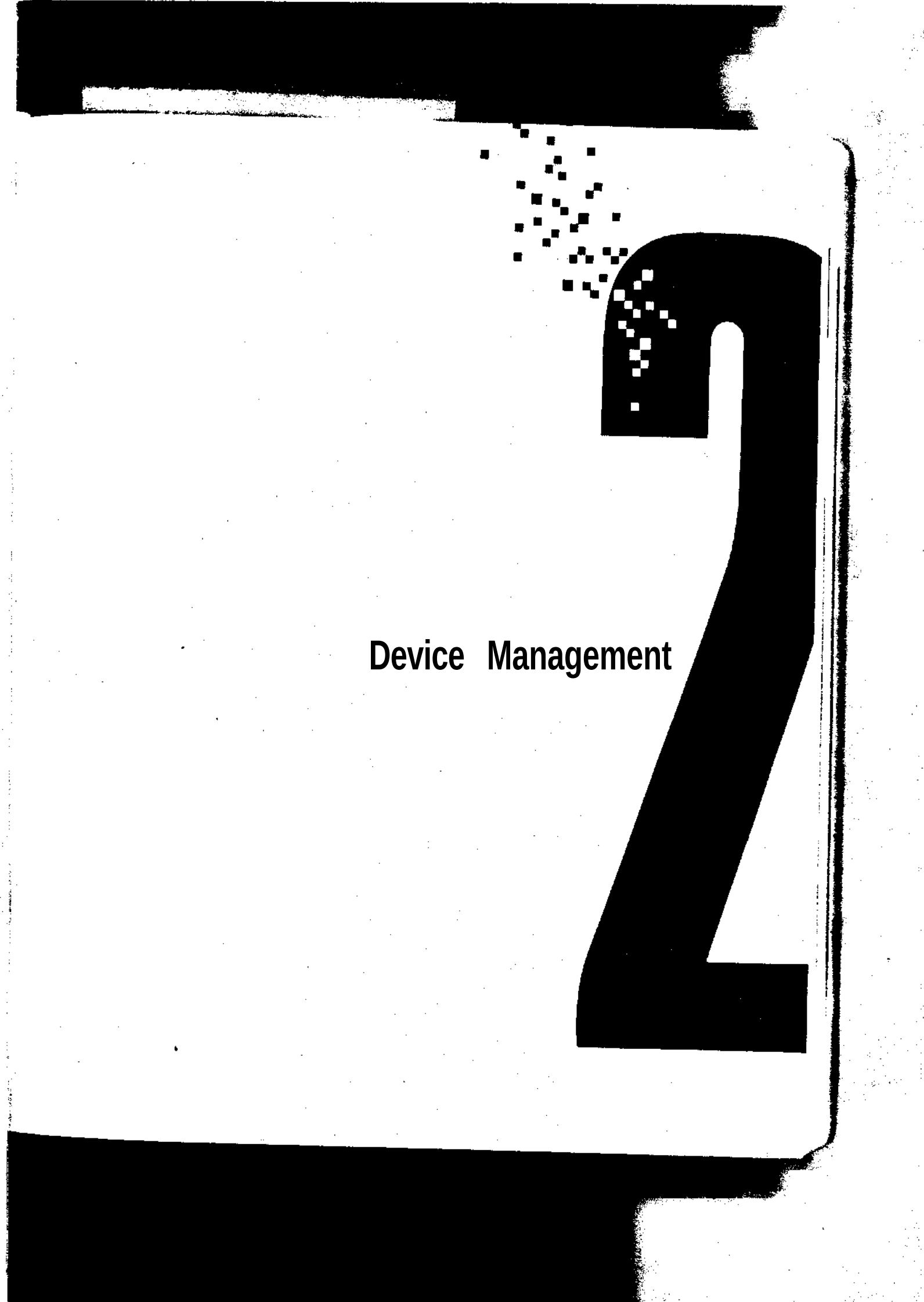
In addition to the I/O request structure, most devices use other structures to help manage the device. The Audio, Parallel, Serial, Timer, and TrackDisk devices have one additional structure each. The Clipboard device has two, the Keyboard device has three, and the Printer device has four additional structures.

Nine devices have one INCLUDE file each that defines their structures and other data required for a task to interface with the device internal routines. However, the Console, Input, Keyboard, and Printer devices have two INCLUDE files each.

If you study the data in Table 1.2, you will know the names of all device-related structures and where to find structure definitions and other data needed to deal with each Amiga device.

Table 1.2: *Device Structures and INCLUDE Files*

Device	Name of Request Structure	Name of I/O Request Substructure	First Auxiliary Structure	Second Auxiliary Structure	Third Auxiliary Structure	Fourth Auxiliary Structure	INCLUDE Files
Audio	IOAudio	IORequest	**AudChannel**	-	-	—	Audio, h
Clipboard	IOClipReq	IOStdReq	Clipboard_ UnitPartial	SatisfyMsg	-	-	Clipboard.h
Console	lOStdReq	—	ConUnit	-	-	-	Console.h Consoleunit.h
Gameport	IOStdReq	-	_	-	-	-	Gameport.h
Input	IOStdReq	-	InputEvent	-	-	-	Input, h Inputevent.h
Keyboard	IOStdReq	_	KeyMapNode	KeyMap	KeyMap-Resource	-	Keyboard.h Keymap.h
Narrator	Narrator_rb Mouth_rb	IOStdReq	-	-	-	_	Narrator.h
Parallel	IOExtPar	IOStdReq	lOPArray	—	—	-	Parallel.h
Printer	lOPrtCmdReq IODRPReq	IOStdReq	PrinterData	PrinterSegment	PrinterEx-tendedData	DeviceData	Prtbase.h Printer.h
Serial	IOExtSer	IOStdReq	IOTArray	—	_	—	Serial.h
Timer	TimerRequest	IORequest	TimeVal	—	—	_	Timer, h
TrackDisk	IOExtTD	IOStdReq	TDV_Public-Unit	-	-	-	Trackdisk.h



7

Device Management

Introduction

This chapter discusses general topics of vital importance to Amiga device management and programming. Programming procedures for Amiga devices differ from input/output procedures for most other computers. These Amiga procedures were designed so that a programmer could take maximum advantage of the built-in Amiga device internal routines.

The chapter first presents the C language programming procedures you will use for Amiga device management. The most common types of device management tasks, the usual sequences of their execution, and the most important steps in their programming procedures are identified. Following sections focus on the AbortIO, BeginIO, RemDevice, and AddDevice functions, since their uses are similar for all 12 Amiga devices. Altogether, these topics establish a programming framework upon which you can build Amiga device management tasks and programs.

The chapter concludes with discussions of the nine Exec-support library functions. All of these functions are contained on disk in the file named amiga.lib on the C language programming disk in the LIB: directory. Each function is a set of prepackaged program statements that is usually repeated again and again in device management programs. These functions were put together in a library for easy programmer access; they allow your programs to be much shorter than they would be if you programmed using the Exec task and message-port management functions. These functions further streamline your use of the preprogrammed device management features, thus saving programming effort. (The appendix presents a precise definition of each of the Exec-support library functions.)

General Programming Procedures

Figure 2.1 depicts the general sequence of programming steps you should follow when programming Amiga devices. This sequence consists of opening the device unit with OpenDevice; sending commands to the device unit with BeginIO, DoIO, or SendIO; and closing the device unit with CloseDevice. The following discussion is presented in terms of a single task, device unit, task reply port, and I/O request structure. The same programming pattern holds for multiple instances of these, as you will see later on.

The programming steps are as follows:

1. Create a task that will handle the device management. You can do this with the CreateTask function. Note that CreateTask is called in another task, not in the task that will manage the device. (All the devices that the system deals with are managed by a set of device management tasks created by other tasks in the system; the original boot task is the master task in the system.) Once the device management task is created, you can name it (specify a Task structure Node substructure ln_Name parameter) and add it to the system task list using the Exec library AddTask function. All other tasks and programs in the system can then obtain a pointer to its Task structure using the Exec library FindTask function.

2. Create a task reply port using the CreatePort function. This is the task message port where I/O request messages will be queued when the device finishes processing each I/O request. This function call is made within the device management task itself. {Note that the device unit also has a message port for I/O requests that are sent to it. This port is controlled by the Unit structure MsgPort substructure created by the OpenDevice function call; a task is not required to create it with a CreatePort function call inside that device management task.) A task should create a message port for each distinct category of I/O requests, which usually means a separate port for each device and device unit.

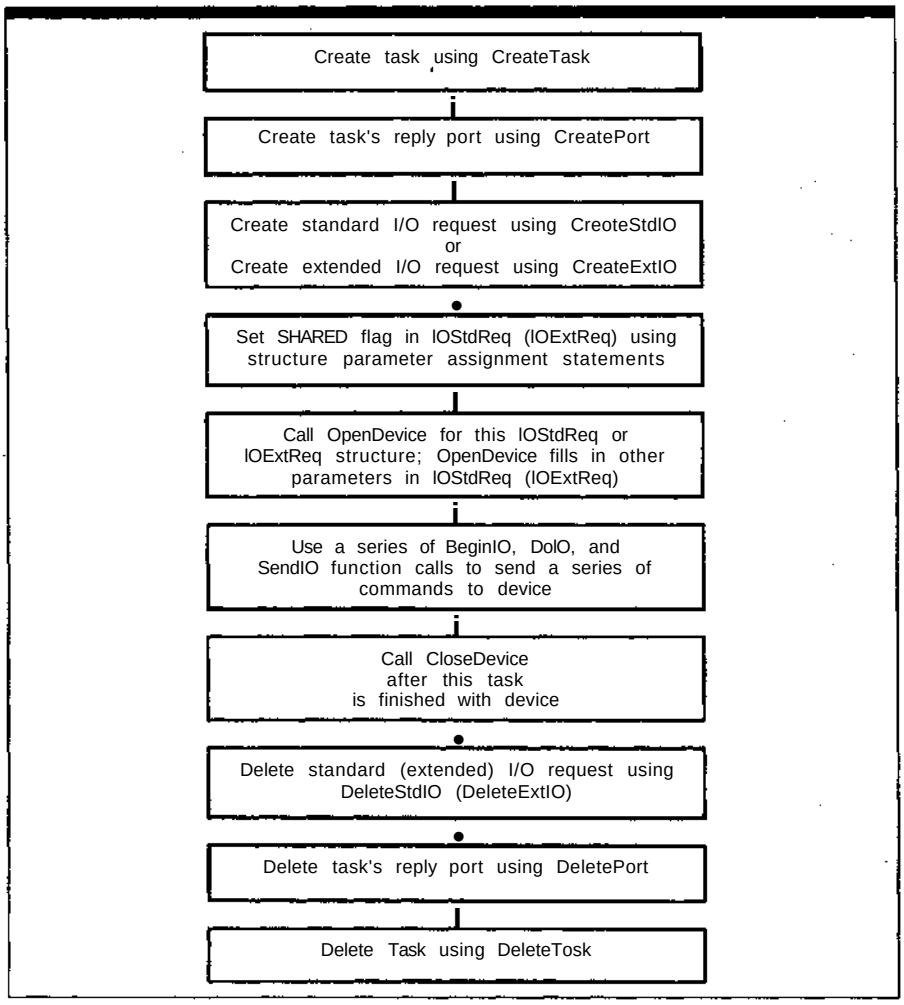


Figure 2.1: *Programming Steps for Task-Device I/O Processing*

3. Create an I/O request structure for the device unit using the CreateStdIO or CreateExtIO function. Use CreateStdIO only if the device requires an IOStdReq structure to define its commands. If the device requires a device-specific I/O request structure, call CreateExtIO to allocate and initialize that structure. A task will usually also have to specify additional structure parameters to fully initialize the device-specific I/O request structure. (See the appropriate chapter to determine what kind of I/O request-structure parameters a device requires.) The I/O request structure is now in the task's memory space, is not queued, and is said to belong to the task.

4. Set the appropriate flag parameter bits in the IORequest, IOStdReq, or device-specific I/O request structure before calling OpenDevice. In particular, decide whether you want to open the device in shared or exclusive access mode. You also may want to set other device-specific I/O request parameters; read the appropriate chapters in this volume to determine what those parameters are.

5. Call the OpenDevice function to open the device unit. OpenDevice will automatically increment the Device structure lib_OpenCnt parameter, indicating that one more task has the device open. It will also automatically increment the Unit structure unit_OpenCnt parameter, indicating that one more task is using the unit. OpenDevice will fill in additional parameters in the I/O request structure; it will set the I/O request structure io_Device and io_Unit parameters to point to appropriate Device and Unit structures.

6. Use a series of BeginIO, DoIO, and SendIO function calls to send a series of commands to the device unit. First map out the task's data needs and decide the order in which the task needs the data. Then decide if the task must have the data before proceeding (synchronous) or can request data and go on to other things (asynchronous). Then send the commands.

7. Close the device unit when you are sure that this task will no longer need it. Generally speaking, call CloseDevice using the same I/O request structure you used when you called OpenDevice. This step will decrement the Device structure lib_OpenCnt parameter, indicating that one less task has the device unit open; it will also decrement the Unit structure unit_OpenCnt parameter, indicating that one less task is using the device unit.

8. Delete the I/O request structure using the DeleteStdIO or DeleteExtIO function. Use DeleteStdIO if you originally used CreateStdIO; use DeleteExtIO if you used CreateExtIO. These function calls free the memory occupied by the I/O request structures.

9. Delete the task reply-port MsgPort structure using the DeletePort function. This frees the memory occupied by the task message-port management structures. However, if you want to use the task message port for other messages in the system, don't delete it at this time.

10. Arrange to return to the task that originally created the device management task, and call the DeleteTask function to delete the Task structure used to manage the device management task. (This is an optional step.)

Asynchronous I/O Request Processing

Figure 2.2 shows how the system behaves while processing asynchronous I/O requests. You use asynchronous I/O requests when you want to send multiple I/O requests one after the other and you do not need the data before you can continue with your task. Recall that five functions—three Exec library and two device library functions—control the detailed operation of asynchronous I/O request processing: BeginIO, SendlO, AbortIO, ChecklO, and WaitiO. In addition, the Exec library GetMsg and Remove functions also play a part in the sequence of actions for asynchronous I/O request processing.

Six rectangles in the figure represent a sequence of actions that are in part directly under programmer-task control, but for the most part are determined and controlled by the coordinated action of the system and device internal routines. For this illustration, QuicklO was not requested.

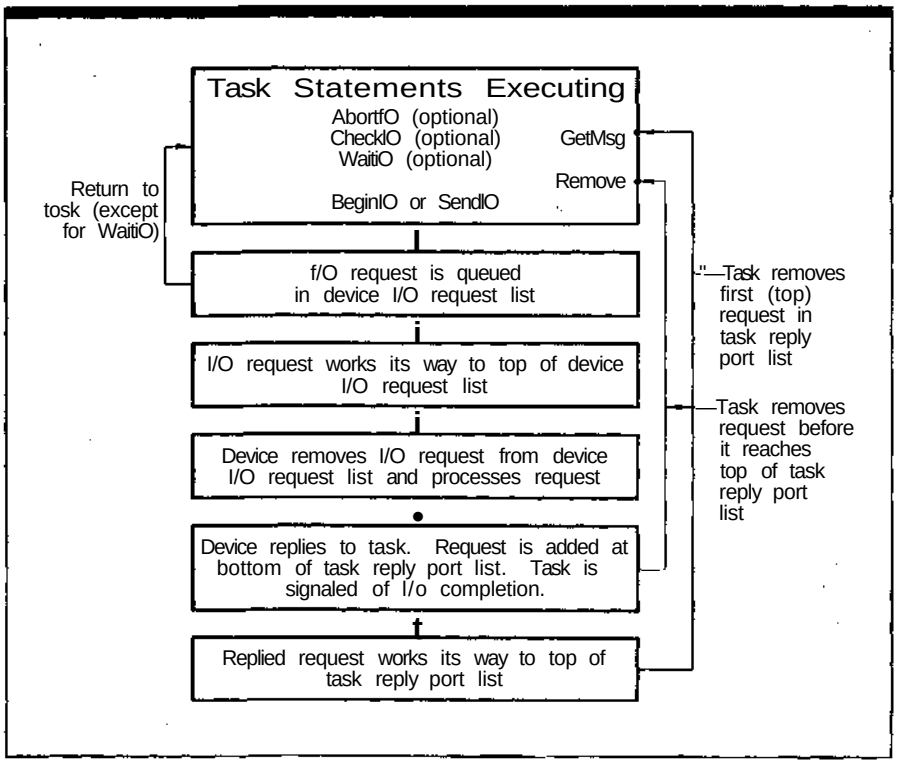


Figure 2.2: *Progress of an Asynchronous I/O Request*

The action proceeds as follows:

1. The task has sent an asynchronous I/O request using either the BeginIO or SendIO function. (Figure 2.1 showed the sequence of steps that led up to this point in the task; note that a particular device unit was previously opened with OpenDevice.)

2. The I/O request is placed in the device-unit request queue using the Device, Unit, MsgPort, and Message structures associated with the queue's message port and the I/O request. The device-unit request queue is a first-in, first-out (FIFO) queue, so this particular I/O request was placed at the bottom of the queue. The I/O request structure is now on the device-unit request queue and is said to belong to the device internal routines. If the task does not execute a WaitIO function call for the I/O request, it can continue executing other task statements. If the task does execute WaitIO at any time after it executes BeginIO or SendIO, and the device has not processed and sent the I/O request back to the task reply-port queue, that task will block further execution and lose the CPU until a task signal indicates the device has returned the I/O request. The mechanism to detect and handle the task reply-port signal must be established before the I/O request is sent.

3. The queued I/O request has now worked its way to the top of the queue and can be processed by the device internal routines. Just how long the I/O request takes to get from the bottom of the list to the top depends on the current length of the list and on all other activities in the system. How busy the system is determines how much time the CPU can give to the device internal routines of this particular device and how fast it can remove I/O requests from its device request queue.

4. The device internal routines remove the pending I/O request from the top of the device-unit request queue. The device uses the GetMsg function internally to get the I/O request. It then uses the parameters in the I/O request structure to determine what the requesting task wants back from the device.

5. The device replies to the task and adds the reply to the bottom of the task reply-port queue. The I/O request structure is now in the task reply-port queue and is said to belong to the task. If an error occurred during processing, the replied I/O request structure will contain information about the error in the io_Error parameter. The I/O request structure Message substructure mn_ReplyPort parameter tells the system where to put the reply; the device uses the ReplyMsg function internally to send the reply to the task reply port. In addition, if a signaling mechanism has been specified between the task reply port and the task that owns it, the task will be signaled of the arrival of the reply.

6. The task, once signaled, can remove the I/O request from the task reply-port queue immediately using the Remove function, as depicted in the figure by the line from the fifth small rectangle to the first one. Remove is usually used only in conjunction with the CheckIO function, which checks for the presence of a specific reply in the queue. CheckIO returns a pointer to the queued I/O request structure, and the task can then call the Remove function to remove it. Note that this can also be

done while the replied I/O request is working its way to the top of the task reply-port queue.

7. If the reply message is not removed from the task reply-port queue using the Remove function, the task will continue processing queued replies on a first-in, first-out basis when it becomes active. The replied I/O request originally sent by BeginIO or SendIO will work its way to the top of the queue, and the task can remove it. To do so, the task must explicitly call the GetMsg function, shown in the figure by the line from the sixth small rectangle to the first.

This completes the travels of the asynchronous I/O request through the system, from the sending task back to the sending task. First sent to a specific device unit by the BeginIO or SendIO function, the request proceeds through the various queues and finally returns to the originating task, where it can be accessed, the returned data processed, and the I/O request reused if necessary.

Remember that the system is generally switching between this task, other programmer-defined tasks, and system-defined tasks. Also, when a device is active, the CPU will spend its time executing the device internal routines. All of these actions take place under the control and supervision of the Exec routines and the combined system and device internal routines. The queues help arbitrate the sequence of events.

Figure
Progress
Synchr
I/O Re

Synchronous I/O Request Processing

Figure 2.3 shows how the system behaves while processing synchronous I/O requests. You usually use synchronous I/O requests when you want to send one request at a time and wait on the device to send the data back to your task. Note that the Exec GetMsg function does not play a part in removing replied synchronous I/O requests from the task reply-port queue. Again, QuickIO was not requested in this example.

There are a few important differences between synchronous and asynchronous processing, as a comparison of the two figures shows. These are the points to remember about synchronous I/O:

- The CheckIO, SendIO, and WaitIO functions are not used. Just like GetMsg, these three functions are only used for asynchronous I/O requests.
- Once the BeginIO or DoIO function executes, the requesting task will be blocked until the I/O request is completed and the requesting task is notified of its arrival in the task reply-port queue. The task will stop execution until the device returns the reply and the task accesses the returned data. In the meantime, while the requesting task is blocked, other tasks may take over the CPU. This is the principal reason why synchronous I/O requests are used.
- BeginIO or DoIO automatically call Remove to remove the I/O request from the bottom of the task reply-port queue as soon as the task is signaled.

Always remember that no I/O request structure in the system is ever copied as it moves from queue to queue during I/O request processing. Instead, only one copy of these structures is present in RAM for each I/O request sent to a device unit. The system manages an I/O request structure by placing it on a set of queues in a well-controlled

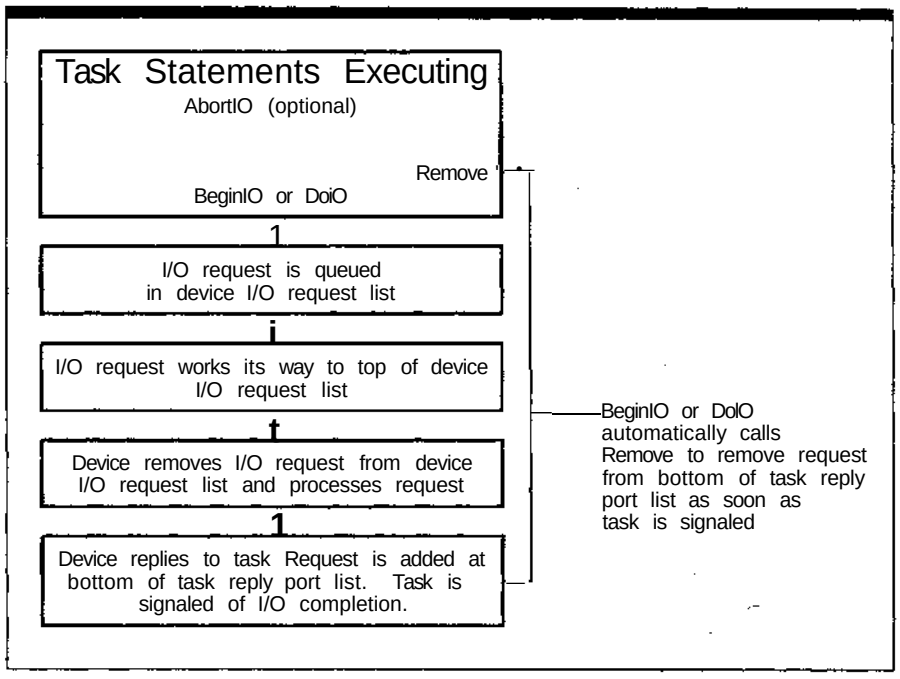


Figure 2.3: *Progress of a Synchronous I/O Request*

sequence. In this way RAM requirements are minimized and each I/O request structure is reusable once it completes a round trip.

Multiple Tasks, Reply Ports, and Device Interaction

Figure 2.4 shows three tasks, each containing some of the Exec-support library function calls as part of its task statements. The figure does not depict the exact order of function call execution; rather, it depicts the general packaging of Exec-support library functions to work with Amiga devices.

Task2 was created by a CreateTask call in Taskl. Task3, in turn, was created by a CreateTask call in Task2. Thus, Task3 also was created indirectly by Taskl.

As an example of how this works, think of Device 1 as the Serial device connected to a modem. Device2 is the Audio device connected to a set of speakers, and Device3 is the Printer device connected to a printer through the Amiga parallel port. Under this arrangement, you would want a Serial device management task, an Audio device management task, and a Printer device management task. Creating three distinct tasks in this way would help you keep things straight in your program.

Depending on your particular program design, the situation could be even more complicated: Taskl needs data from Device 1 in two different categories, Task2 needs data from Device2 in two different categories, and Task3 needs data from Device3 in two different categories. In order to make data management easier in this situation, it is necessary to create two types of I/O request structures and two types of I/O request reply ports for each task.

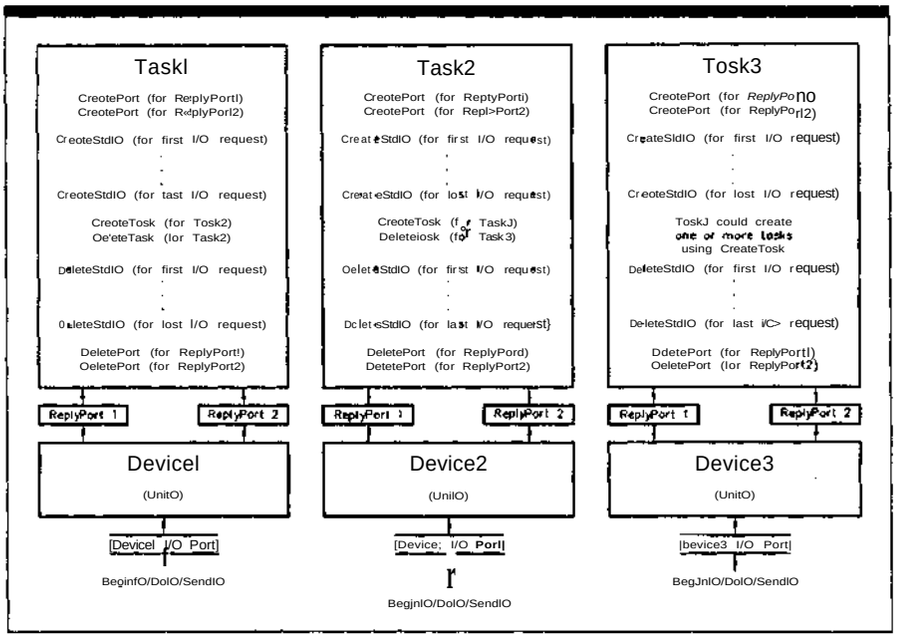


Figure 2.4: ***Using the Exec-Support Library Functions for Tasks and Reply Ports***

Taskl contains two CreatePort statements to create two task reply ports for I/O requests coming back from Devicel. In addition, Taskl contains a number of CreateStdIO function calls to create IOStdReq structures that send commands to Devicel, one CreateStdIO function call to create an IOStdReq structure to use when calling OpenDevice, one CreateStdIO function call for each command to be passed, and a corresponding number of task-cleanup DeleteStdIO function calls to delete the IOStdReq structures. Finally, Taskl contains a CreateTask function call to create Task2 and a task-cleanup DeleteTask function call to delete Task2. The same pattern of function-call packaging holds for Task2 and Task3.

Immediate-Mode Request Processing

This section discusses the concept of immediate-mode request processing. Because immediate-mode commands operate in a unique way, they play an important role in Amiga device management.

CMD_CLEAR, CMD_FLUSH, CMD_START, CMD_STOE, and CMD_RESET can be dispatched as immediate-mode commands. Moreover, these commands are not *always* dispatched as immediate-mode commands; they often operate in other modes, depending on the predefined characteristics of the device internal routines.

Consider the various ways that a device command can progress through the Amiga device software system. Taking into account that both the device request queue and the task reply-port queue are possible holding locations for I/O requests, there are four possible

paths an I/O request can follow:

1. A command can be dispatched to a specific device unit and queued in its I/O request queue; the device can then process the command and reply it to the task reply-port queue. This path involves queuing on both ends of the transaction.

2. A command can be sent to a device unit with the I/O request structure io_Flags IOF_QUICK bit set. QuicklO will be successful if the current system conditions allow it. The device internal routines wili process the command immediately and send it back to the task without queuing on either end of the transaction. The IOF_QUICK bit will still be set in the replied I/O request.

3. A command with IOF_QUICK set can be sent to a device unit, but the device internal routines may not be able to process the command as a QuicklO command. They will then automatically queue the command I/O request in the device-unit request queue. It will be processed when it reaches the top of the queue and will be subsequently replied to the task reply-port queue by the device internal routines. This arrangement also involves queuing on both ends of the transaction. The IOF-_QUICK bit will be reset in the replied I/O request.

4. A command can be sent **to** the device **unit in** immediate mode. The device internal routines always process it immediately, no matter what else is going on in the system at that time. These commands have high priority; they may or may not be replied to the task reply-port queue, depending on the IOF_QUICK bit setting.

As an example, assume that a task has sent an immediate-mode CMD_RESET command to reset a device's internal routines to their default startup state. The task is asking that the device internal routines be reset immediately, regardless of what circumstances currently exist in the system. It is easy to see that such a command should not be queued in the device request queue.

Moreover, the task usually wants the device internal routines to execute the command immediately, even if it involves interrupting a currently executing nonimmediate command. This is best illustrated by the example of a CMD_STOP command interrupting a CMD_READ or CMD_WRITE command. In many cases, the CMD_READ or CMD_WRITE command will be interrupted in the middle of its progress, thereby leaving some bytes not yet transferred. The device internal routines will not allow the data transfer to complete before CMD_STOP stops the device from sending data back and forth between the device and task-defined buffers.

The precise operational details of the immediate-mode commands vary from device to device. They can sometimes be dispatched as QuicklO and can sometimes be replied to the task reply-port queue. The appropriate command discussions in the following chapters detail the specific behavior of individual commands.

General I/O Request-Structure Procedures

This section discusses procedures for initializing and dealing with I/O request structures. You can apply these procedures to programming any of the 12 Amiga devices.

Because the IORequest structure is the minimum structure required to dispatch I/O requests to device units, specialized device-specific I/O request structures always include it as the first substructure entry. Therefore, any parameters in the IORequest structure are also in device-specific structures.

The IORequest structure consists of a Message substructure containing an mn_RepIyPort parameter; the io_Device and io_Unit pointer parameters; and the io_Command, io_Flags, and io_Error parameters. Here are the procedures for initializing these parameters regardless of the device you are programming:

- mn_ReplyPort. This parameter must usually be initialized to point to the MsgPort structure representing the task reply port of the task sending the I/O request. When a device has finished processing a command, its internal routines will send the reply to this port. Those routines will place a pointer to the I/O request structure on a list that represents all queued I/O request structures that have come back from the device unit. Note that the reply mechanism is handled automatically by the device internal routines. The mn_ReplyPort parameter can also point to *any* MsgPort structure belonging to *any* task in the system.

  Specifying mn_ReplyPort to point to a MsgPort structure owned by a task other than the one that originated the I/O request provides a mechanism for the originating task to send device data to another task. This is another mechanism for indirect data transfer between tasks using the device internal routines to generate that data. If you think of the device internal routines as a third task in this process, you have one task sending data to another task using a third task to generate that data.

  If you specify a null value for mn_ReplyPort, the device internal routines will not reply the I/O request to any task reply-port queue. In most cases, the originating task will not be able to get an I/O request structure pointer and will therefore not be able to retrieve the device-generated data.

  If a task uses the CreateStdIO or CreateExtIO functions, the mn_ReplyPort argument is the only argument in the CreateStdIO function call and the first of two arguments in the CreateExtIO function call. In addition, CreateStdIO and CreateExtIO initialize the I/O request structure Message substructure ln_Type and ln_Pri parameters; therefore, if a task uses these functions to create its I/O request structures, it will not have to initialize these two parameters.

- io_Device. This is a pointer to a Device structure used to manage the device internal routines of a specific device. It is initialized by the OpenDevice call for the first I/O request structure used by a task to communicate with a specific device unit. Additional I/O request structures that the task needs in order to send commands to the device can then be initialized by copying this parameter into their io_Device parameters. The same Device structure is used for all open units of a specific device.

- io_Unit. This is a pointer to a Unit structure used to establish a specific device-unit I/O request-queue message port and to manage a particular device unit. It is also initialized by the OpenDevice call for the first I/O request structure used by a task to communicate with a specific device unit. Additional I/O request structures

that the task needs to send to the device unit can then be initialized by copying this parameter into their io_Unit parameters.

- io_Command. This is a literal constant representing the name of one of the device commands. The INCLUDE files assign a specific value to it for every device command that a particular device honors. You always initialize this I/O request-structure parameter using a simple C language structure-parameter assignment statement.

- io_Flags. This represents a set of flag parameter bits representing the specific requirements of the I/O request a task sends to a device unit. In some cases a task initializes this parameter before it opens a device unit with OpenDevice. For example, if a task needs to open a device unit in shared access mode, it should initialize the io_Flags parameter shared access bit in the first I/O request structure before calling OpenDevice. Some devices are opened in exclusive access mode unless a task specifies otherwise in the io_Flags parameter. Other flag parameter bits should also be set in other device-specific OpenDevice calls; in the following chapters, you will discover the flag parameter bits that are provided for each specific device.

  Once you have defined the first I/O request structure to open the device, you may want to set the io_Flags parameter to other values for other I/O requests. For example, if a device supports QuicklO, a task can initialize io_Flags to IOF-_QUICK for some (or all) commands sent to that device.

- io_Error. The value of this parameter is usually set by the device internal routines before the I/O request is replied.

Classes of I/O Requests

Ail I/O requests fall into two general classes:

1. Those defined by an IOStdReq structure. The IOStdReq structure consists of an IORequest substructure with four parameters (io_Actual, io_Length, io_Data, and io_Offset) appended to it. The io_Data parameter enables a task to point to a RAM data area (a task-defined buffer) that can be used as a source and a destination for information coming from and going to the device internal routines. (The IORequest substructure itself has a total of six parameters but does not include any parameters to represent a RAM data-area pointer; by itself, it does not allow a task and device to relate through task-defined buffers.)

2. Those defined by a device-specific extended I/O request structure. An example is the TrackDisk device, which uses the IOExtTD structure to manage data going back and forth between a task and a specific device unit. All of the device-specific extended I/O request structures are summarized in Table 1.3 in Chapter 1.

Creating Multiple I/O Requests

Each I/O request sent to a device requires a distinct I/O request structure to represent it. »A specific I/O request structure must not currently be on any list in the system if you

want to use it; it cannot be in a device request queue or a task reply-port queue. This section presents the rules you should follow when creating multiple I/O request structures to define the data needs of your tasks.

A task can get its required I/O request structures in three ways: it can create them anew with a call to OpenDevice; it can reuse already defined ones by redefining their parameters once they have completed a round trip from task to device and back to the task; or it can create new ones by cloning (copying) some parameters and initializing others in an already existing I/O request structure. The procedure a task should use depends on the specific point in that task, what I/O request structures are already defined at that point, and what the task is trying to accomplish.

If an I/O request structure is created anew by an OpenDevice call, OpenDevice first initializes the io_Device and io_Unit parameters to point to a Device and a Unit structure. OpenDevice defines these two parameters for the first usage of the I/O request structure, which represents the first data request actually made to the device unit using BeginIO, DoIO, or SendlO. Once these two parameters are initialized, a task can copy them into any number of other properly allocated I/O request structures. Each of these copy operations, together with other structure-specific parameters, will result in a unique instance of the I/O request structure in RAM.

In addition, the specific I/O request structure initialized by OpenDevice can be used again and again for BeginIO, DoIO, and SendlO function calls, provided it has completed a round trip (from the initializing task to the device-unit request queue, to the task reply-port queue, and back to the task). Once back in the task it will not be on any lists in the system. Only then can the device data represented by the structure be accessed by the task and the I/O request structure be reinitialized and dispatched again. When the task once again owns that particular I/O request structure, its parameters (io_Flags, io_Data, io_Length, and so on) can be redefined and it be can used to send a new I/O request.

Figu

Mul
Reque

Processing Multiple I/O Requests

Figure 2.5 shows a device management task in action. This figure represents the task-device interaction of any of the 12 Amiga devices. For example, the large rectangle could represent the program statements of a disk-data management task handling I/O between a specific TrackDisk device unit and a set of task-defined buffers. The figure will first be discussed in terms of asynchronous I/O requests.

Each of the small rectangles inside the larger one represents a specific instance of an I/O request structure and the task operations required to define it. Each of thes^ I/O request structures could be created by the Exec-support library functions Cre-aceStdIO and CreateExtlO. Then only the I/O request structure Message substructure paraaieters would be initialized; a task would still have to initialize some parameters (io_Data. io__Length, io_Actual, io_Offset, and so on). These I/O request structures could be created the hard way, using individual Exec library functions and assignment statements, but usins Exec-support library functions is more efficient. In either case, the I/O request structure is allocated in RAM at a location determined by the I/O request structure allocadon process. (The figure is not intended to portray the RAM location of these structures. >

The initialization of IORequestO is completed with a call to the OpenDevice function, which initializes its io_Device and io_Unit parameters. Other required IORequescO rxarsmeters

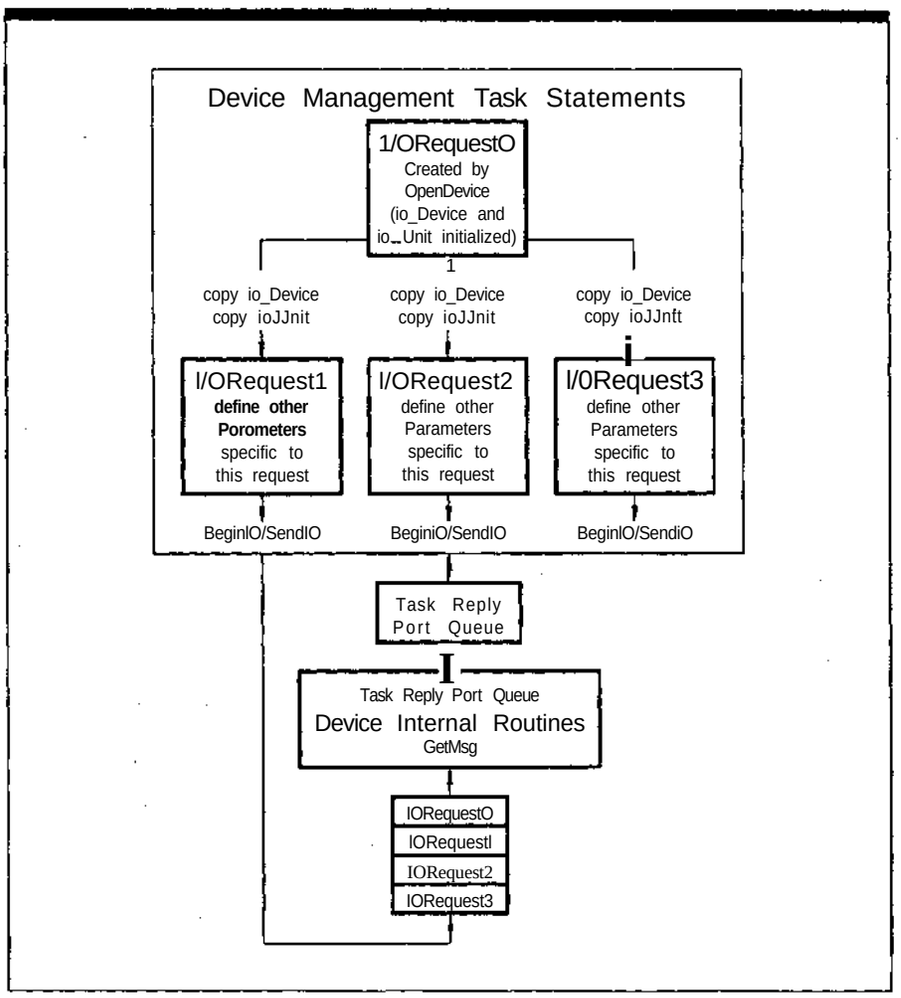


Figure 2.5: *Sending Multiple I/O Requests to a Device*

can be initialized using simple structure-parameter assignment statements. When all of these parameters are initialized, IORequestO is dispatched by a BeginIO or SendIO function call, denoted by the arrow on the right side of the topmost small rectangle.

If the device unit currently has no queued I/O requests, the dispatched request will be queued at the top of the device-unit request queue. On the other hand, if the device unit already has queued I/O requests, IORequestO will be placed below them. The device internal routines will process IORequestO when it gets to the top of the queue. However, if the system is busy with other tasks, devices, or interrupts, IORequestO may have to wait in the queue until the system passes control to the device internal routines.

Once the task has dispatched IORequestO as an asynchronous request, it can go on to other things. It may want to dispatch other I/O requests to the same unit of the same

device. However, it cannot use IORequestO to do so, because the device itself now owns IORequestO as it sits in the device-unit request queue.

The task must therefore create a series of other I/O requests to satisfy its other data needs. These are created by allocating additional I/O request structures and initializing them properly according to the task's data needs. In the figure, IORequestl, IORequest2, and IORequest3 represent both the I/O request structures and the operations required to define them. The structures are created in the usual C language way—with CreateStdIO, CreateExtIO, and structure-parameter assignment statements. However, two of their structure parameters (io_Device and io_Unit) are copied from IORequestO, as shown by the lines between the small rectangles. Copying ensures that the additional I/O request structures will be managed by the Device and Unit structures created by OpenDevice for IORequestO.

Since these are asynchronous I/O requests, the task does not have to sleep after BeginIO or SendIO executes. Moreover, BeginIO and SendIO can work with ChecklO and WaitIO to allow the task to handle replied asynchronous I/O requests. For example, if the task wanted to load four different disk-resident files into RAM from the same physical disk unit, four different I/O request structures could be created and dispatched to the TrackDisk device internal routines. A task could use the TrackDisk device IOExtTD structure to define the details of these requests. Once the TrackDisk device was opened by OpenDevice, a task could create the I/O request structures one after the other, using an io_Device and io_Unit parameter copying operation and initializing other parameters as appropriate in each I/O request structure. Each fully defined I/O request structure could then be dispatched with BeginIO or SendIO.

The task could then go on to other computations and activities not requiring the data in these files. At any point in the sequence of task statements where the task needed the file data, it could check or wait (using ChecklO or WaitIO) for the return of these requests to one of its task reply-port queues. Once they arrived and were removed or moved to the top of that queue, the task could get the file data and continue with its operations.

On the other hand, if the task required the data from a disk file and could not go on to do anything else until it had the file data in one of its task-defined buffers, it would use the DoIO function to dispatch the I/O request to the device unit. Once again, the specific I/O request structure could be copied from another already replied or newly created I/O request structure; but the task would go to sleep until the device sent the file-data reply to the task.

Note that this discussion has focused on one task and one device unit. It can easily be extended to multiple tasks, multiple reply ports, multiple devices, and multiple device units. With due attention to using the proper io_Unit parameter, the procedures for creating and copying I/O request structures and their parameters in a more complex multiunit situation are virtually the same.

Device Library Functions

Five functions are used to control most device I/O operations: AbortIO, BeginIO, ChecklO, DoIO, and SendIO. Although these functions all end with "10," they fall into

two distinct categories: CheckIO, DoIO, and SendIO are Exec library functions; AbortIO and BeginIO, however, are device library functions—they are defined separately in each device library. Although they are part of the device-specific internal routines, they are accessed directly as functions with arguments, just like CheckIO, DoIO, and SendIO.

Since AbortIO and BeginIO occur in each device-specific library and their internal definitions are similar from device to device, this section presents a discussion of the common features and uses of the two functions. Any device-specific differences from this general discussion will be noted in the following chapters.

The AbortIO Function

AbortIO aborts a specified device I/O request after it has been sent to a specific unit of any of the 12 Amiga devices. AbortIO is capable of aborting both active requests and currently queued requests. If the I/O request is queued, it is removed from the device-unit I/O request queue. The I/O request structure representing the device command is then replied to the requesting task's reply-port queue. If the I/O request is currently active, execution of its device command is stopped at the earliest possible moment. The I/O request structure for that command is then replied to the task reply-port queue.

In both cases, the I/O request io_Error parameter is set to IOERR_ABORTED. The task that originally dispatched these requests can then look at the replied I/O request structure and take further action based on the io_Error parameter. In particular, the sending task can modify the I/O request structure as needed and dispatch it again.

In contrast to the CMD_FLUSH command, the AbortIO function provides a mechanism to abort a single I/O request that a task previously placed in the device-unit I/O request queue.

The BeginIO Function

BeginIO enables a task to send a command to the device internal routines of any of the 12 Amiga devices. The device internal routines then look at global conditions in the system to determine how the device command will be processed. BeginIO recognizes other current I/O demands on the device and will process I/O requests according to specific preassigned priority rules.

The operation of the BeginIO function differs from DoIO and SendIO in that it allows a task to send a device command either synchronously or asynchronously. BeginIO is often used in lieu of DoIO for synchronous I/O request commands or SendIO for asynchronous I/O request commands. All 12 Amiga devices allow the use of BeginIO.

Generally speaking, commands dispatched with BeginIO are treated asynchronously or synchronously depending on these considerations:

- The particular device to which the command was dispatched.
- The particular command dispatched to that device.
- Systemwide hardware and software conditions at the time the task dispatches the command.

Once the system selects synchronous or asynchronous command execution, other things happen in a specific order that is uniform for all devices.

If the system executes the command synchronously, it effectively calls the DoIO command to dispatch the command just as if the task used DoIO explicitly. Recall that DoIO puts the sending task to sleep, waiting for the data to come back from the device.

The system also examines the device I/O request structure io_Flags parameter IOF_QUICK bit to see if it was set by the dispatching task. If IOF_QUICK was set, the device internal routines will try to complete the I/O request and send the results to the task using the usual procedures for QuickIO. If QuickIO is successful, the reply will not be sent to the task reply-port queue; instead, the requesting task will get the device data back immediately.

If the task did not set IOF_QUICK, the system will queue the request in the device-unit request queue, and it will be processed when it reaches the top of the queue. It will then be replied to the task reply-port queue. The reply will work its way to the top of that queue; the task will then get the device data, and the loop will be complete.

The sending task can always check to see if QuickIO was successful by looking at the io_Flags parameter. If the IOF_QUICK bit is still set when the I/O is completed, it means that QuickIO was successful The sending task should use CheckIO to check for the return of the QuickIO request.

If the system decides to execute the command asynchronously, the command will be dispatched just as if SendIO was called directly. The sending task will not be put to sleep; it can go on to do other things. In the meantime, the device internal routines first check to see if the task set the I/O request-structure io_Flags IOF_QUICK bit. If it did so, the bit is first cleared and the I/O request is placed in the device-unit request queue. Note that this bit was not necessarily cleared when the system decided to execute the command synchronously.

When the I/O request works its way to the top of the device-unit request queue, the device internal routines process that request. Then they send the reply to the task reply-port queue. Once the I/O request works its way to the top of that queue, the task can get the device data and the loop will be complete.

Remember that for any device command that has a built-in QuickIO capability, the programmer can always set the I/O request structure io_Flags IOF_QUICK bit. However, that command may not be executed as QuickIO—for example, if the system was busy with lots of device activity. The system software and device internal routines can decide that the command must be executed asynchronously.

BeginIO is used most often to dispatch Audio device commands. The Audio device is the most complicated Amiga device because of multitasking, allocation, and arbitration complexities; with multiple tasks all trying to use the same four audio channels, BeginIO provides a valuable predefined internal decision-making mechanism to guide the flow of events.

The RemDevice and AddDevice Functions

This section extends the discussion of the Exec library RemDevice and AddDevice functions presented in Volume I. These functions interact with the Exec library OpenDevice and CloseDevice functions and the individual device library Expunge routines, to manage

the memory resources assigned to devices. The Expunge routine is built into the device internal routines for all 12 Amiga devices. A C language task does not call Expunge directly but indirectly through the RemDevice function.

A task calls the AddDevice function to add a device to the system device list. Once added, any task in the system can refer to that device by name. The system automatically adds at least the Input, Console, Timer, and TrackDisk devices to the system device iist upon startup. In addition, a task can add any of the other Amiga devices to the system device list with explicit calls to AddDevice. A device then remains on the list until it is removed explicitly by RemDevice or until it is removed indirectly by a CloseDevice function call following a RemDevice call.

A direct call to RemDevice attempts to both remove the device from the system device list and expunge the specified device from the system, thereby freeing memory resources for other tasks and uses. In particular, RemDevice attempts to free the RAM assigned to the Device and Unit structures and other memory assigned to the device. To accomplish this, it calls the specific device library Expunge routine.

The exact results achieved by a RemDevice call depend on the prior history of device management when RemDevice is called. However, one rule is certain: RemDevice will not be immediately successfúl unless all device units, once opened with OpenDevice calls, have subsequently been closed with CloseDevice calls. Remember that each OpenDevice and CloseDevice call opens and closes one device unit. In an OpenDevice call, the specified unit is indicated as one of the function call arguments. In the CloseDevice call, the specified unit is indicated by the specified I/O request structure io_Unit parameter. Therefore, although a device is always open when any of its units are open, it is not fully closed until all of its units are closed.

If a task calls RemDevice for a specific device when any of its device units are still open, the system will not expunge the device immediately but will instead set a device structure parameter for a deferred expunge. The bookkeeping required for this scheme is maintained by the system using two parameters in the Device and Unit structures assigned to each device unit. The system automatically increases the Device structure lib_OpenCnt and the Unit structure unit_OpenCnt parameters by 1 each time OpenDevice is called for any device unit. It automatically reduces the Device structure lib_OpenCnt parameter and the Unit structure unit_OpenCnt parameter by 1 each time CloseDevice is called for any device unit.

With this arrangement, if the Device structure lib_OpenCnt and Unit structure unit_OpenCnt parameters are not both 0 when RemDevice is called, the RemDevice expunge operation will be deferred until all tasks that currently own a device unit close those device units with CloseDevice. If a task calls RemDevice while any device unit is still open, the system automatically sets the Device structure lib_Flags parameter to LIBF_DELEXR, thus indicating a pending deferred expunge. The deferred expunge will actually take place when the last task currently having a device unit open closes that device unit.

Once all units of a device have been closed and the device has been removed from the current system device list by a successful RemDevice call (perhaps due to a deferred expunge), no new OpenDevice calls for any device unit will succeed; any task that wants to open a device unit must first call the Exec library AddDevice function to add that device to the system device list.

The interactions of the AddDevice, RemDevice, OpenDevice, and CloseDevice functions get more complicated if multiple device units are opened in shared access mode among a set of tasks sharing units of a device. These considerations are discussed in the OpenDevice and CloseDevice function discussions in following chapters.

USE OF EXEC-SUPPORT LIBRARY FUNCTIONS

CreateExtIO

Syntax of Function Call

iORequest = CreateExtIO (iOReplyPort, size)

Purpose of Function

This function allocates and initializes an IOExtReq structure and sets the Message substructure reply-port pointer parameter, mn_ReplyPort$_3$ to the value specified by the iOReplyPort argument. An IOExtReq structure is an extended device-specific structure whose size varies from device to device.

CreateExtIO returns a pointer to an IORequest structure for the I/O request a task will use to send a command to a device. That IORequest structure is always a substructure in an extended I/O request structure that is used for a specific type of device.

Inputs to Function

iOReptyPort	A pointer to a MsgPort structure representing the reply port where I/O request replies should be sent when the device internal routines have finished processing them
size	The size of the extended I/O request structure in bytes

Discussion

Two functions in the Exec-support library deal with extended I/O request structures: CreateExtIO and DeleteExtlO. You can use the CreateExtIO function to allocate and initialize device-specific extended I/O request structures in your tasks. The Serial and Parallel devices are examples of devices that use extended I/O request structures.

Because the IORequest structure is the first entry in the extended I/O request structure, a pointer to it is also a pointer to the extended I/O request structure. Each device that does not use the IOStdReq structure has its own extended I/O request structure; its size depends on the data needs of that device. All of these device-specific I/O request structures are defined in the Amiga INCLUDE files (see Table 1.3). A task can use the C language sizeof operator to determine the number of bytes required for any extended I/O request structure. To pass a new command to a device requiring an extended I/O request structure, a task should first create a new extended I/O request structure for that command.

CreatePort

Syntax of Function Call

msgPort = CreatePort (msgPortName, msgport_priority)

Purpose of Function

This function declares and initializes a MsgPort structure with a specified name and priority. It allocates a signal bit number for a signal to be assigned to this message port. CreatePort also adds the message port to the system message-port list using the msgport_priority argument to fix the requested position in that list.

The msgPortName argument provides a way for other tasks to rendezvous with (obtain a pointer to) this message port; any task can use the FindPort or FindName function to get a pointer to the MsgPort structure for this message port by using its name as the input argument. CreatePort returns a pointer to a MsgPort structure for the newly created message port. This MsgPort structure is used to define and control the message port while it is active in the system.

Inputs to Function

msgPortName	A pointer to a null-terminated string representing the name of the message port you want to create; the MsgPort structure Node substructure ln_Name parameter is then set to this value
msgport_priority	The list-position priority (-128 to 127) that you want to assign to this message port in the system message-port list; the MsgPort structure Node substructure Jn_Pri parameter is set to this value

Discussion

Two functions in the Exec-support library deal with message ports: CreatePort and DeletePort. You use CreatePort to create, allocate, and initialize message ports for your tasks. These message ports can be used to queue any messages in the system, no matter where they originate. Moreover, if you are programming devices, these ports can act as task reply ports for the I/O requests sent back to your task by any device unit in the system.

Note that the MsgPort structure required by the device-unit internal routines is defined and managed by the Unit structure (discussed in Chapter 1). Every task in the system that exchanges information with that device unit automatically queues its I/O requests in that unit's I/O request list. The CreatePort function is used to create the task reply-port I/O request-queue message port, not the device-unit I/O request-queue message port. Always keep this distinction in mind.

You can create any number of task reply ports (limited, of course, by available RAM). In addition, the use of CreatePort and DeletePort is not restricted to device management tasks; you can use them to create and delete any message ports (and reply ports) in your programs, no matter what type of messages you are passing between any two tasks in the system.

CreateStdIO

Syntax of Function Call

iOStdReq = CreateStdIO (iOReplyPort)

Purpose of Function

This function declares and initializes an IOStdReq structure and sets the reply-port pointer parameter (mn_ReplyPort) in its Message substructure to the value specified by the ioReply-Port argument. CreateStdIO also sets the IOStdReq structure Node substructure ln_Pri parameter to 0, indicating that the I/O request should be placed at the bottom of the task reply-port queue when it is replied by the device internal routines.

CreateStdIO returns a pointer to an IOStdReq structure. A task can use this structure to send any command to any device unit for which the IOStdReq structure is the required I/O request structure.

Inputs to Function

iOReplyPort	A pointer to a MsgPort structure that represents a task reply port

Discussion

Two functions in the Amiga system deal with IOStdReq structures: CreateStdIO and DeleteStdlO. You can use the CreateStdIO function to allocate and initialize IOStdReq structures in your tasks. Each time a task needs to pass a new command to a device, that task should create a new IOStdReq structure (or reuse a previously replied I/O request structure) for that command.

CreateTask

Syntax of Function Call

taskCB = CreateTask (taskName, task_priority, taskEntryPoint, task_stack_size)

Purpose of Function

This function declares and initializes a Task structure and sets the task priority to the specified task_priority argument using the Task structure Node substructure ln_Pri parameter. CreateTask also establishes the RAM entry point for initial task execution, initializes the Task structure stack control parameters (tc_SPReg, tc_SPUpper, and tc_SPLower), and adds the task to the system task list using the task__priority argument to determine the position in that list.

Once all of this is done, CreateTask returns a pointer to the Task structure for the newly created task. This Task structure is used to control the task while it is active in the system.

Inputs to Function

taskName	A pointer to a null-terminated string representing the name of the task; the Task structure Node substructure ln_Name parameter is set to this value
task_priority	The task priority (a value from -128 to 127) of the newly added task
taskEntryPoint	A pointer to the task RAM entry point; it is used as the initPC parameter in the AddTask function calf inside the CreateTask function definition

task_stack_size The size of the RAM stack assigned to this task; it is used by the CreateTask function to establish the stack control parameters

Discussion

Two functions in the Exec-support library are used to manage tasks—CreateTask and DeleteTask. They control the allocation and deallocation of RAM and signals, and other bookkeeping operations required to keep track of the resources used by a task.

Use of the CreateTask and Delete Task functions is not restricted to device management tasks; you can use these two functions to create and delete any tasks in the system, regardless of how those tasks will be used. They do other things as well; see the appendix, which presents the actual C language definition of these functions.

DeleteExtIO

Syntax of Function Call

DeleteExtIO (iOExtReq, size)

Purpose of Function

This function deallocates the memory for an extended I/O request structure originally allocated by CreateExtlO. DeleteStdIO sets the extended I/O request-structure Node substructure ln_Type parameter to a hexadecimal FF value and decrements the IOStdReq structure io_Device and io_Unit parameters by 1.

Inputs to Function

iOExtReq A pointer to a device-specific extended I/O request structure; usually the pointer originally returned by the CreateExtlO function

size The size of the extended I/O request structure as defined in CreateExtlO

Discussion

Two functions in the Exec-support library deal with extended I/O request structures: CreateExtIO and DeleteExtIO. DeleteExtIO deallocates all memory assigned to an extended I/O request structure. You can use DeleteExtIO to delete any extended I/O request structure from the system; it does not have to originate with CreateExtIO.

DeletePort

Syntax of Function Call

DeletePort (msgPort)

Purpose of Function

This function deletes the specified MsgPort structure from the system message-port list and deallocates the memory originally allocated by the CreatePort function for that MsgPort structure.

DeletePort also sets the MsgPort structure Node substructure ln_Type parameter to a hexadecimal FF value and reduces the mp_MsgList List structure lh_Head parameter by 1, indicating one less message port on the system message-port list. Finally, DeletePort calls the Exec FreeSignal funaion to free the signal bit number assigned to the message port by the CreatePort function.

Inputs to Function

msgPort	A pointer to a MsgPort structure; usually the pointer originally returned by the CreatePort function

Discussion

Use the DeletePort function to deallocate the RAM originally allocated by the CreatePort function for the MsgPort structure. Note that any task can call the DeletePort function to delete any message ports from the system; that message port does not have to originate with the CreatePort function. All a task needs is a pointer to a MsgPort structure, no matter how that MsgPort structure was created in the first place.

DeleteStdIO

Syntax of Function Call

DeleteStdIO (iOStdReq)

Purpose of Function

This function deallocates the memory originally allocated for an IOStdReq structure by the CreateStdIO function. It also decrements the IOStdReq structure io_Device and io_Unit parameters by 1 and sets the IOStdReq structure Node substructure in_Type parameter to hexadecimal FF.

Inputs to Function

iOStdReq	A pointer to an IOStdReq structure; usually the pointer returned by CreateStdIO when the IOStdReq structure was originally allocated and initialized

Discussion

You can use the DeleteStdIO function to deallocate the memory originally allocated to an IOStdReq structure by the CreateStdIO function. CreateStdIO and DeleteStdIO do other things as well; see the appendix.

Note that any task can use DeleteStdIO to delete any IOStdReq structure from the system; the structure does not have to originate with the CreateStdIO function.

DeleteTask

Syntax of Function Call

DeleteTask (taskCB)

Purpose of Function

This function deallocates the memory originally assigned to the Task structure by the CreateTask function. It frees the Task structure RAM (places it on a memory-free list) so that another task can use that memory for its needs. DeleteTask also removes the task from the system task list using the Exec library RemTask function.

Inputs to Function

taskCB — A pointer to a Task structure used to control the task while it is active in the system; usually the pointer originally returned by CreateTask

Discussion

CreateTask and DeleteTask should be used as a pair; doing so provides a great deal of convenience to the programmer. Just as you allocate and initialize the Task structure with the CreateTask function, you use the DeleteTask function to deallocate the Task structure from the system.

NewList

Syntax of Function Call

NewList (list)

Purpose of Function

This function initializes a new list in the system by calling the assembly language NEWLIST macro.

Inputs to Function

list — A pointer to a List structure that will control a new list in the system

Discussion

The NewList function calls the assembly language NEWLIST macro. Once the List structure is created, nodes can be added to or deleted from the list by defining appropriate Node structures.

For an example, look at the definition of the CreatePort function in the appendix. The NewList function is used to create a new list for messages in a message port if the msgPortName pointer argument in the CreatePort function is 0, indicating that the new message port is unnamed.

Introduction

The Console device is used to send data to an Intuition window or to receive input from the Amiga keyboard, gameport connectors, or disk drives. It also opens the Input device automatically, which in turn opens the Keyboard, Gameport, and Timer devices automatically. Input events coming to the Console device usually originate with these other devices. The Console device is one of the input handlers that process input events. It is positioned at priority 0 in the input-handler function list described in Chapter 7.

Unlike other Amiga devices, the Console device does not work with the Unit structure; instead, it works with the ConUnit structure, which enables a task to represent a connection to the Console device internal routines through its MsgPort substructure and a connection to an Intuition window and the Intuition internal routines through its cu_Window parameter.

Operation of the Console Device

Figure 8.1 shows the general operation of the Console device. A task can read characters from the Amiga keyboard while also writing ASCII characters and screen control characters to an Intuition window. The figure shows how a task receives information from the disk system, the Amiga keyboard, and the mouse, and where that^ information goes.

A Console device unit is automatically associated with an Intuition window by the OpenDevice call that opens it. OpenDevice initializes a ConUnit structure to tie together the MsgPort structure and the Intuition Window structure. The device-unit request queue is managed by the ConUnit structure MsgPort substructure, and each Intuition window is managed by an Intuition Window structure. The ConUnit structure is the medium of communication between the Console device internal routines and the Intuition internal routines.

A task that needs to use the Console device internal routines to process mouse, keyboard, or gameport input events should establish a separate task reply-port queue for queuing replied CMD_READ commands. In order for the Console device to communicate with an Intuition window, it should also establish a separate task reply port for queuing replied CMD_WRITE commands. In addition, if the task needs to dispatch any other Console device commands, it should set up a third task reply port. The Exec-support library CreatePort function should be used to create these ports.

Read-Write Operations for the Console Device

Figure 8.2 illustrates the general operation of the Console device for write operations. Each CMD_WRITE command dispatched to the Console device internal routines sends either a set of ASCII characters or a set of screen control characters to a Console device unit Intuition window.

Each open Console device unit is always tied to an Intuition window, which acts like an enhanced ASCII terminal. It obeys many of the standard ANSI screen control-code (escape-character) sequences, as well as additional sequences unique to the Amiga. The

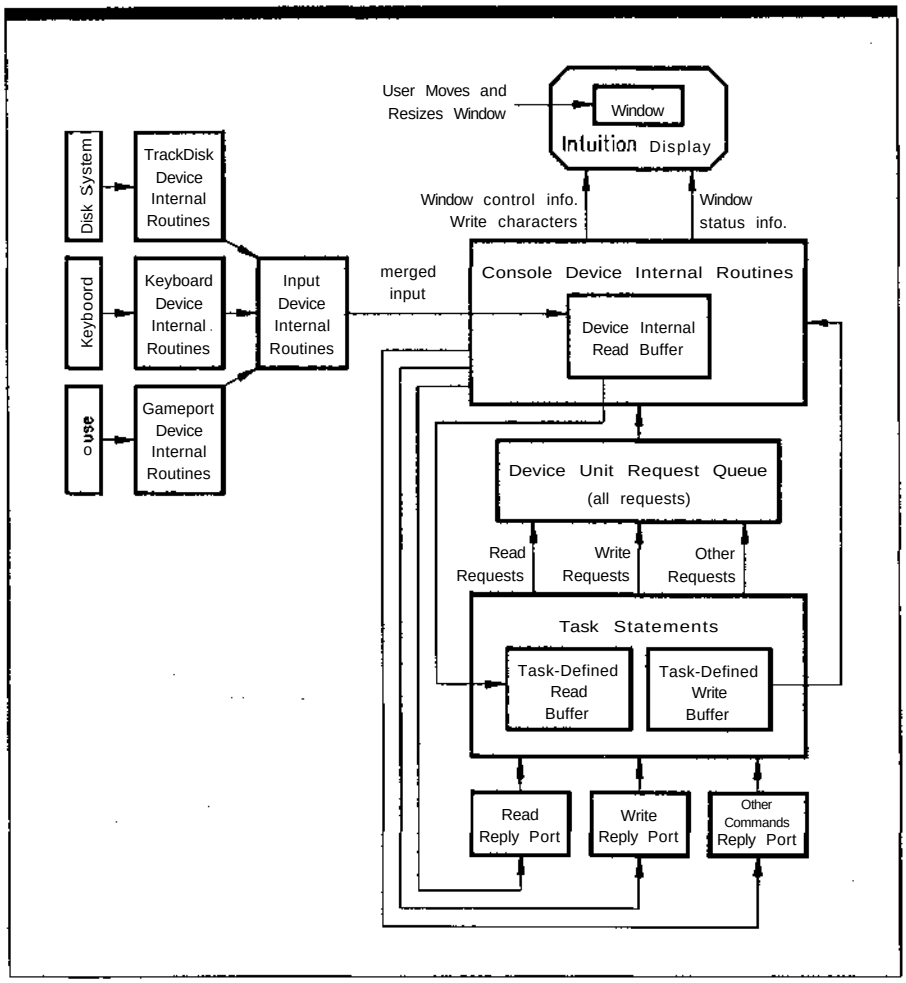


Figure 8.1: *Operation of the Console Device*

open Console device unit can also send an ASCII character stream to its associated Intuition window, which becomes the text the user sees in the window. It is the responsibility of each task to define the information in its task-defined write buffers before a CMD_WRITE command is dispatched.

The relationship between the Console device internal routines and the Intuition internal routines is shown in the lower half of Figure 8.2. The set of arrows between the ConUnit structure and the Intuition internal routines represents the information transfer path.

Each Console device unit Intuition window is positioned initially with its upper-left corner at pixel coordinates (11,11). The Intuition software system internal routines keep track of the continuously changing size and location of each window and automatically supply and update that information in a set of 14 ConUnit structure parameters.

Figure 8.2 *Writ Operation for th> Consol Devic*

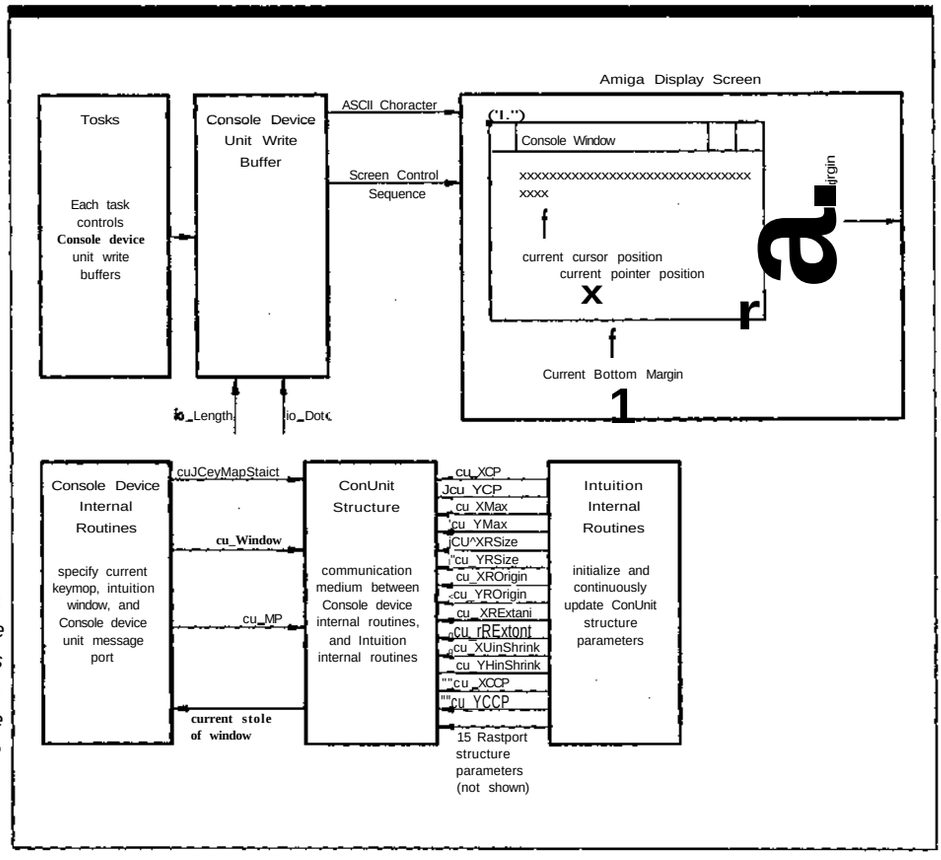


Figure 8.2: *Write Operations for the Console Device*

This allows a task and the Console device internal routines to monitor the current state of the window. These parameters are in addition to the 15 RastPort structure parameters that the Intuition system initializes and updates. A task can read the 14 ConUnit structure parameters but cannot write (change) them. The Console device internal routines use these parameters to determine how to place text into the Intuition window. Window manipulations by the user are often the cause of parameter changes.

The Console device routines receive information from the task through the message port defined by the ConUnit structure cu_MP MsgPort substructure, which represents the message port where a task can queue CMD_READ, CMD_WRITE, and other I/O requests for processing by the Console device internal routines. The ConUnit cu_Window parameter is provided as input to the OpenDevice function call when the Console device unit is first opened. The cu_KeyMapStruct parameter represents the name of the current KeyMap structure used for key mapping during CMD_READ processing.

Figure 8.3 illustrates the general operation of the Console device for read operations. 'Each CMD_READ command dispatched to the Console device routines reads one of the

following into a task-defined read buffer from the Console device internal read buffer:

- A continuous byte stream of ANSI 3.64 characters coming from,the Amiga keyboard indirectly through the Keyboard device and Input device internal routines. This stream may contain ASCII characters or raw input event information.
- A continuous mouse input stream coming from the Amiga mouse indirectly through the Gameport and Input device internal routines because of a user's mouse actions in an Intuition window.
- A disk insertion or removal input event coming from the TrackDisk device and Input device internal routines indirectly when a user changes the disk in a disk drive.

The Console device can deal with two kinds of input events: raw and preprocessed. A music program, for example, may want to deal with keyboard input events as raw events, with no keymapping or raw code translation, whereas a text program may want to deal with keyboard input events after they have been preprocessed into ASCII and escape-character sequences.

All three categories of input events can be either raw or preprocessed, depending on the setting of the SRE (set raw events) and RRE (reset raw events) parameters. Each input event was originally represented as an InputEvent structure; the input events were merged by the Input device routines before reaching the Console device. See Chapter 7 for further details on this operation.

The event stream coming from the keyboard can be preprocessed by a key map before its individual characters are sent to the Console device internal read buffer. The key map changes (maps) each character into another character or string of characters; these are then read into the task-defined buffer for further processing. The system provides a default key map (standard United States), or a user can define one. The Keyboard device system provides the KeyMap structure and the CD_ASKKEYMAP and CD_SETKEY-MAP functions to manage the key map.

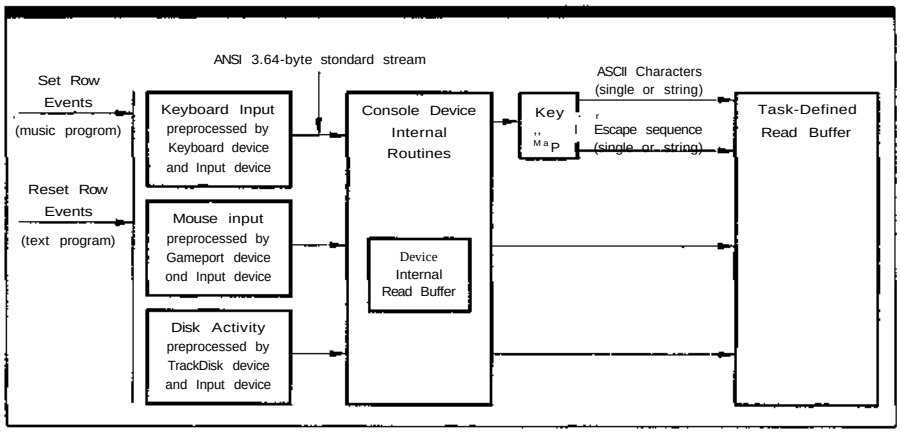


Figure 8.3: *Read Operations for the Console Device*

The Amiga currently supports the following built-in key maps: German, Spanish, French, British, Italian, Icelandic, Swedish/Finnish, Danish, Norwegian, French Canadian, and standard United States. It also supports a Release 1.1-compatible standard United States key map and a key map that changes the keyboard from a standard Qwerty to a Dvorak keyboard. These key maps are in the Workbench disk's system directory. The user selects the Setmap icon and makes an Info menu selection to choose a specific key map.

Once the character stream is preprocessed by the key map, it is divided into two— one character stream consisting of standard ASCII characters (either singly or in a string), and the other a set of characters defining an escape-character sequence, which is either a single charaaer or a string of characters preceded by the ASCII escape character. Both of these character streams are placed into the Console device read buffer for processing by a task. For example, the characters can be sent to an Intuition window by dispatching an appropriate CMD_WRITE command that uses the task-defined read buffer as a write buffer for the characters.

Mouse input events are sent directly from the Gameport device internal routines to the Console device routines for processing. These input events lead to a set of actions in the Intuition window. Disk insertion and removal input events are sent directly from the TrackDisk device routines to the Console device routines for processing. These events can lead to a set of actions in the Intuition window (for example, a requestor that tells the user to insert a specific disk).

Console Device Commands

The Console device has four device-specific commands and three standard device commands. All commands support both QuicklO and queued I/O. No command supports immediate-mode operation. All commands affect the IOStdReq structure io_Error parameter; CMD_READ also affects the io_Actual parameter and the contents of the Console device internal read buffer.

Sending Commands to the Console Device

Figure 8.4 depicts the general scheme used to dispatch commands to the Console device internal routines. The lines with arrows represent the parameters you initialize and those returned by the Console device internal routines. The individual function and command sections in this chapter indicate the appropriate parameters for your task.

The programming process consists of three phases:

1. IOStdReq structure preparation. The programmer has complete control over this phase; here, you initialize parameters in the IOStdReq structure in preparation for dispatching a command to the Console device internal routines. The parameters include the usual ones required by most devices, as well as arguments for the CDInputHandler and RawKeyConvert functions; the choice of parameters depends on the specific command or function you plan to dispatch. These parameters provide an information path to the data needed by the Console device internal routines to process the command or function.

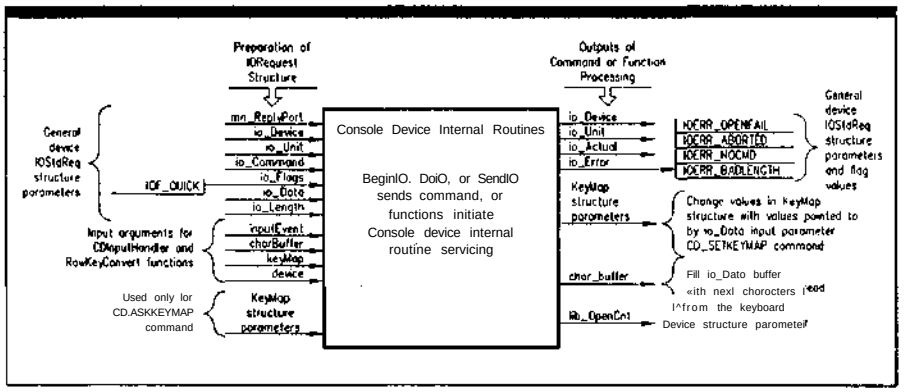


Figure 8.4: *Console Device Command and Function Processing*

2. Console device routine processing. The only part you play in this phase is to dispatch the command to the device using BeginIO, DoIO, or SendIO. When one of these functions begins executing, control passes to the device and system internal routines.

3. Command output parameter processing. The system and Console device internal routines have complete control over this phase. The results of Console device command processing have been returned to the task that originally dispatched the command. If the I/O request was not successful as QuickIO, it was processed when it moved to the top of the device-unit request queue; the Console device then replied and the request is now in the task reply-port queue. If the request was a successful QuickIO, it was not queued in the task reply-port queue but came directly back to the requesting task; the four parameters still direct you to appropriate data for your task.

For most of the Console device commands, the system provides the io_Error output parameter; for CMD_READ it also provides the io_Actual output parameter. In addition, the CD_ASKKEYMAP and CD_SETKEYMAP commands read or write key map data into the KeyMap structure.

Note that Figure 8.4 also shows the parameters that play a part in Console device function setup and processing. The OpenDevice and CloseDevice functions both affect the unit 0 Device structure lib_OpenCnt parameter; OpenDevice also affects the io_Error parameter. Note that the ConUnit structure does not contain an open-count parameter equivalent to the Unit structure unit_OpenCnt parameter used with other devices.

Structures for the Console Device

Figure 8.5 illustrates the structures required to define operations for the Console device. The Console device deals directly with only one structure—ConUnit. However, the Console device also requires three other structures to work with the Amiga keyboard: KeyMapNode, KeyMap, and KeyMapResource.

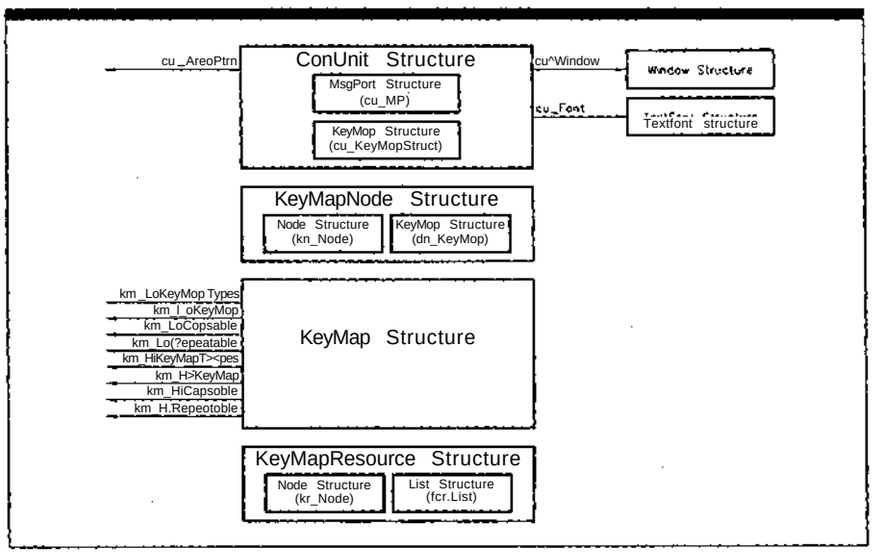


Figure 8.5: *Console Device Structures*

The ConUnit structure contains two substructures, a MsgPort structure named cu_MP and a KeyMap substructure named cu_KeyMapStruct. The MsgPort structure is discussed in Chapter 1 of Volume *I;* the KeyMap structure is discussed in Chapter 9 of this volume.

ConUnit also contains three pointer parameters. The cu_AreaPtrn parameter points to a Graphics library drawing-area pattern in RAM (see Volume I, Chapter 2); cu_Window points to an Intuition Window structure (see Volume I, Chapter 6); and cu_Font points to a TextFont structure (see Volume I, Chapter 4). These parameters help manage the Intuition window associated with the Console device unit.

The KeyMap structure contains no substructures, but it does contain a set of eight pointer parameters. Each points to a different area of RAM in which information for a specific key map is kept.

The KeyMapNode structure contains two substructures, a Node substructure named kn_Node and a KeyMap structure named kn_KeyMap. The system uses the Node structure to place each KeyMap structure on a system list of KeyMap structures.

The KeyMapResource structure contains two substructures, a Node structure named kr_Node and a List structure named krJList. The system uses them to maintain a list of current keyboard resources in the system.

The ConUnit Structure

The ConUnit structure is defined as follows:

```
struct Con Unit {
    struct MsgPort cu_MP;
    struct Window *cu_Window;
```

```
    WORD cu_XCP;
    WORD cu_YCP;
    WORD cu_XMax;
    WORD cu_YMax;
    WORD cu_XRSize;
    WORD cu_YRSize;
    WORD cu_XROrigin;
    WORD cu_YROrigin;
    WORD cu_XRExtant;
    WORD cu_YRExtant;
    WORD cu_XMinShrink;
    WORD cu_YMinShrink;
    WORD cu_XCCP;
    WORD cu_YCCP;
    struct KeyMap cu_KeyMapStruct;
    UWORD cu_TabStops[MAXTABS];
    BYTE cu_Mask;
    BYTE cu_FgPen;
    BYTE cu_BgPen;
    BYTE cu_AOLPen;
    BYTE cu_DrawMode;
    BYTE cu_AreaPtSz;
    APTR cu_AreaPtrn;
    UBYTE cu_MinTerms[8];
    struct TextFont *cu_Font;
    UBYTE cu_AlgoStyle;
    UBYTE cu_TxFlags;
    UBYTE cu_TxHeight;
    UBYTE cu_TxWidth;
    UWORD cu_TxBaseline;
    UWORD cu_TxSpacing;
    UBYTE cu_Modes [(PMB_AWM + 7)/8];
    UBYTE cu_RawEvents[(IECLASS_MAX + 7)/8J;
};
```

The cu_MP parameter is the MsgPort substructure representing the Console device-unit request queue; cu_Window points to an Intuition Window structure representing the window associated with the unit. The next 14 parameters are task read-only parameters. They are initialized when OpenDevice returns and are kept up-to-date automatically by the Intuition software system internal routines to reflect changing conditions in the Intuition window:

- cu_XCP and cu_YCP are the current X and Y positions of the last character placed into the Intuition window.
- cu_XMax and cu_YMAX are the current maximum allowed X and Y positions of a character in the Intuition window.

- cu_XRSize and cu_YRSize are the maximum number of characters that can be placed into the window in the X and Y directions. These parameters are used for automatic word wrap and for line formatting in the window.
- cu_XROrigin and cu_YROrigin are the X- and Y-direction origins of the Intuition window associated with the Console device unit.
- cu_XRExtant and cu_YRExtant are the current maximum X- and Y-direction sizes of the window raster associated with the Console device unit.
- cu_XMinShrink and cu_YMinShrink are the current minimum X- and Y-direction sizes allowed for the Intuition window after the user (or a task) resizes the window.
- cu_XCCP and cu_YCCP are the current X and Y of the cursor in the window. They change as the user moves the cursor.

The next two parameters in the Con Unit structure can be read and written to (changed) by a task:

- cu_KeyMapStruct is the name of a KeyMap substructure used by the Console device unit for mapping keystrokes. The KeyMap structure can be changed by the AskKeyMap and SetKeyMap functions.
- cu_TabStops[MAXTABS] is a set of longwords representing the current tab stops in the Intuition window.

The next 15 parameters are the Con Unit structure values for the Graphics library RastPort substructure used to control the drawing of graphics and text into the Intuition window. Read Chapter 2 of Volume I to see how these parameters are defined and used. Following is a brief summary:

- cu_Mask is the RastPort structure write mask parameter.
- cu_FgPen is the RastPort structure foreground pen parameter.
- cu_BgPen is the RastPort structure background pen parameter.
- cu_AOLPen is the RastPort structure area-outline pen parameter.
- cu_DrawMode is the RastPort structure drawing-mode parameter.
- cu_AreaPtSz is the RastPort structure area-pattern size parameter.
- cu_AreaPtrn is the RastPort structure area-pattern parameter.
- cu_MinTerms[8] is the RastPort structure minimum terms parameter.
- cu_Font is a pointer to a TextFont structure associated with the RastPort structure.
- cu_AlgoStyle is the RastPort structure algorithimic style parameter.
- cu_TxFlags is the RastPort structure text flags parameter.

- cu_TxHeight is the RastPort structure text height parameter.
- cu_TxWidth is the RastPort structure text width parameter.
- cu_TxBaseline is the RastPort structure text baseline parameter.
- cu_TxSpacing is the RastPort structure text-spacing parameter.

The last two parameters to the ConUnit structure are for system use only:

- cu_Modes [(PMB_AWM + 7)/8)] is a set of eight Console device unit modes. Each bit in this byte parameter represents one mode. The parameter is used internally by the Console device routines.
- cu_RawEvents[{IECLASS_MAX+7)/8] is a set of raw event switches. This number is tied to the maximum number of raw event classes; it is used internally by the Console device routines.

The KeyMap Structure

The KeyMap structure is defined as follows:

```
struct KeyMap {
    UBYTE  *km_LoKeyMapTypes;
    ULONG  *km_LoKeyMap;
    UBYTE  *km_LoCapsable;
    UBYTE  *km_LoRepeatable;
    UBYTE  *km_HiKeyMapTypes;
    ULONG  *km_HiKeyMap;
    UBYTE  *km_HiCapsable;
    UBYTE  *km_HiRepeatable;
};
```

The parameters in the KeyMap structure have the following meanings:

- km_LoKeyMapTypes points to the type of translation table to be used for key mapping; in this case, the table that covers the raw key codes from hexadecimal 00 through 3F.
- km_LoKeyMap points to a translation table that defines a translation for raw key-code values between hexadecimal 00 and 3F. Each entry in this table is four bytes long. The translation table can generate a single character or a string of characters for each raw key code. Values for the space bar, the Tab, Alt, Ctrl, and arrow keys, and several other keys are not included here; they are included in the high-key map table.
- km_LoCapsable points to an 8-byte table (64 bits) containing more information about the raw key-code translation process; it tells the system how to treat the Shift and Caps Lock key status. The table represents keys whose raw key codes are between hexadecimal 00 and 3F. The bits that control it are numbered from bit 0

in byte 0 to bit 7 in byte 7 in linear fashion; for example, the bit representing the capitalization status for the key transmitting raw key code 00 is in bit 0 in byte 0.

- km_LoRepeatable points to an 8-byte table (64 bits) that tells the system if the specified key should repeat when pressed. The table represents keys whose raw key codes are between hexadecimal 00 and 3F. The bits that control this feature are again numbered from bit 0 in byte 0 to bit 7 in byte 7 in linear fashion.
- km__HiKeyMapTypes points to the type of translation table to be used for key mapping; in this case, the table that covers raw key codes from hexadecimal 40 through 67.
- km__HiKeyMap points to a translation table that defines a translation for raw key-code values between hexadecimal 40 and 67. Each entry in this table is four bytes long. The table can generate a single character or a string of characters for each raw key code. Values for the space bar, the Tab, Alt, Ctrl, and arrow keys, and several other keys are included in this table.
- km_HiCapsable points to an 8-byte table (64 bits) containing more information about the raw key-code translation process; it tells the system how to treat the Shift and Caps Lock key status. The table represents keys whose raw key codes are between hexadecimal 40 and 67. The bits that control it are numbered from bit 0 in byte 0 to bit 7 in byte 7 in linear fashion; for example, the bit representing the capitalization status for the key transmitting raw key-code 40 is in bit 0 in byte 0.
- km_HiRepeatab!e points to an 8-byte table (64 bits) that tells the system if the specified key should repeat when pressed. The table represents keys whose raw key codes are between hexadecimal 40 and 67. The bits that control this feature are again numbered from bit 0 in byte 0 to bit 7 in byte 7 in linear fashion.

The KeyMapNode Structure

The KeyMapNode structure is defined as follows:

```
struct KeyMapNode {
    struct Node kn_Node;
    struct KeyMap kn_Keymap;
};
```

The parameters in the KeyMapNode structure have the following meanings:

- kn_Node is the name of a Node substructure used to place a set of KeyMap structures on a list.
- kn_Keymap is the name of the KeyMap structure to be placed on the KeyMap structure list.

The KeyMapResource Structure

The KeyMapResource structure is defined as follows:

```
struct KeyMapResource {
   struct Node kr_Node;
   struct List kr_List;
};
```

The parameters in the KeyMapResource structure have the following meanings:

- kr_Node is the name of a Node substructure used to place a set of KeyMapNode structures on a list.
- kr_List is the name of a List substructure used to hold the list of KeyMap structures.

USE OF FUNCTIONS

CDInputHandler

Syntax of Function Call

```
newInputEvent = CDInputHandler (oldInputEvent, device)
D0                              A0             A1
```

Purpose of Function

This function handles input events for the Console device. The ROM input task is usually responsible for producing input events; the CDInputHandler function processes some of them. Input events not processed by CDInputHandler are passed on to one of the Input device's input-handler functions.

CDInputHandler returns a pointer to an InputEvent structure in the newInputEvent variable, which points to the first of a group of one or more input events that were not processed by the CDInputHandler function. Each of these input events is also linked with the InputEvent structure ie_NextEvent parameter; the list of input events is then sent to the Input device handler functions for further processing.

CDInputHandler is included in Release 1.2 to ensure compatibility with programs that may have used it before Release 1.2 was available. A Release 1.2 program should not use the CDInputHandler function; instead, it should use the input-handler functions associated with the Input device, as described in Chapter 7.

Inputs to Function

oldInputEvent A pointer to an InputEvent structure representing the first input event in a linked list

device A pointer to a Device structure

Preparation of the IOStdReq Structure

Initialize mn_ReplyPort to point to the MsgPort structure representing the desired task reply port. Initialize io_Device and io_Unit to point to the Device and ConUnit structures that manage unit -1 of the Console device. These parameters can always be copied from the IOStdReq structure initialized by an OpenDevice function call.

Discussion

CDInputHandler is the only Console device function that directly processes input events. It works with the linked list of InputEvent structures. The InputEvent structure ie_NextEvent parameter links the InputEvent structures together; all InputEvent structures in the list are not necessarily in contiguous RAM, so the ie_NextEvent pointer parameter allows the task to link them properly. The entire list of input events is passed to the CDInputHandler function for processing. Input events that are not processed by the CDInputHandler are then sent to the Input device input-handler functions. The newInputEvent parameter returned by CDInputHandler points to the first InputEvent structure in the shortened linked list.

CloseDevice

Syntax of Function Call

CloseDevice (iOStdReq)
A1

Purpose of Function

This function closes access to a specific Console device unit. If this is the last CloseDevice function call for all Console device units in the task and the Input device has also been closed, the Timer, Keyboard, and Gameport devices will also be closed. When CloseDevice returns,

the task cannot use the specific Console device unit until it executes another OpenDevice function call for that unit. CloseDevice sets the IOStdReq structure io_Device and io_Unit parameters to - 1; a task cannot use that IOStdReq structure again until these parameters are reinitialized by OpenDevice. It also reduces the Device structure lib_OpenCnt parameter by 1 to indicate that one less task is using the Console device unit.

Inputs to Function

iOStdReq A pointer to an IOStdReq structure

Discussion

CloseDevice terminates access to a set of device routines for a specific Console device unit and its associated Intuition window. When a task is done with its Console device operations for a specific Intuition window, it should close the unit associated with that window with a call to CloseDevice. This frees memory that might be needed by the system for this or other tasks. Then another task can open, use, and close the Console device for that window; the sequence can be repeated in a C language program that uses Console device routines.

A task should always verify that all of its Intuition window I/O requests have been replied by the Console device routines before it calls CloseDevice. It can do so by using the GetMsg, Remove, CheckIO, and WaitIO functions to see what requests are currently in the task reply-port queue.

The last CloseDevice function call in a task automatically closes the Input, Timer, Keyboard, and Gameport devices in that task. However, since the Timer and Keyboard devices are shared access mode devices, they can remain open in other tasks that have either opened them explicitly or opened them indirectly through the Input device or the Console device.

OpenDevice

Syntax of Function Call

```
error = OpenDevice ("console.device", unit, iOStdReq, 0)
D0                 A0                   D0   A1        D1
```

Purpose of Function

This function opens access to the internal routines of the Console device. OpenDevice also opens the Input device, which in turn opens the Timer, Gameport, and Keyboard devices if they have not already been opened in the current task.

If unit -1 is specified, the OpenDevice call simply gets a pointer to a Device structure that the CDInputHandler and RawKeyConvert functions can use to reach the Console device internal routines. If unit 0 is specified, a Console device unit will be associated with an Intuition window. Unit 0 is used for all Intuition windows that the task wants to associate with a Console device unit.

The OpenDevice function automatically initializes a ConUnit structure to manage the newly opened Console device unit; it contains a MsgPort substructure representing the device request queue for that unit, as well as a pointer to a Window structure representing the associated Intuition window. OpenDevice also increments the Device structure lib_OpenCnt parameter by 1, indicating that one more task has opened the Console device.

The Console device routines assume that the Intuition library and window are already open before OpenDevice is called. As part of the OpenDevice function call preparation, the IOStdReq structure io_Data parameter must be initialized to point to an Intuition Window structure that will represent the window. The RastPort structure associated with the window (see Volume I, Chapter 6) may already be in use by other tasks when the Console device unit becomes associated with the window.

A Console device unit can only be opened in exclusive access mode—it is associated with only one Intuition window. However, the Console device internal routines are always shared among all tasks and units.

The results of function execution are as follows:

- io_Device. This points to a Device structure that manages unit -1 or 0 of the Console device once it is opened.
- io_Unit. This points to a ConUnit structure used to define and manage a MsgPort and Intuition Window structure for Console device unit 0. The MsgPort structure represents the unit 0 device request queue. OpenDevice will assign each newly opened Console device unit a unique ConUnit structure.
- io_Error. A 0 value indicates that the requested open succeeded. IOERR_OPENFAIL indicates that the Console device could not be opened; this is usually caused by a lack of memory.

Inputs to Function

"console.device"	A pointer to a null-terminated string representing the name of the Console device
unit	The Console device unit number
iOStdReq	A pointer to an IOStdReq structure
0	Indicates that the flags argument is ignored

Preparation of the IOStdReq Structure

Initialize mn_ReplyPort to point to a MsgPort structure for the task reply port. Initialize all other parameters to 0, or copy them from an IOStdReq structure for a previous OpenDevice call. Set io_Command to 0_3 or set it to CMD_WRITE or CMD_READ if the task should open the Console device and dispatch a CMD_WRITE or CMD_READ I/O request immediately.

If the CreateStdIO function is used to create the IOStdReq structure, it will automatically return a pointer to an IOStdReq structure; for the Console device, no typecasting is necessary.

Discussion

The OpenDevice function can be called with appropriate parameters to open the Console device and to initialize parameters to define a CMD_READ or CMD_WRITE command. Once a task has opened the Console device, it can dispatch a series of these commands (with BeginIO, DoIO, or SendIO) to send information back and forth between the task, the Amiga keyboard, and the screen display within an Intuition window. Once a task has finished all of its Console device writing and reading, it can (but need not) close the Console device.

Most of the the IOStdReq structure parameters can be initialized after the Console device is open to represent CMD_READ, CMD_WRITE, and other Console device commands. Any parameters that are not explicitly initialized will retain their previous values or be initialized to the default values assigned by the Console device internal routines.

RawKeyConvert

Syntax of Function Call

```
numChars = RawKeyConvert (inputEvent, bufferPointer, bufferLength,
DO                       AO           A1            D1

                         keyMap)
                         A2
```

Purpose of Function

This function converts (decodes or maps) raw key codes into ANSI 3.64-byte values. The conversion is based on the KeyMap structure specified as part of the input definition of the RawKeyConvert function. RawKeyConvert is always called for Console device unit -1.

Recall that the OpenDevice function returns an IOStdReq structure io_Device pointer if the unit-number argument is -1. RawKeyConvert needs this value to obtain a pointer to the Device structure that manages the Console device internal routines. In this way, the Console device internal routines can obtain a function vector offset to the RawKeyConvert function. The CDInputHandler function works in the same way.

The results of RawKeyConvert execution are found in the io_ActuaI parameter, which contains the actual number of ANSI byte charaaers placed into the buffer. If the IOStdReq structure io_Length parameter is not given a high enough value, the io_Actual parameter will be -1, indicating a buffer overflow condition. In this case, not all of the ANSI byte characters in the buffer will necessarily be valid; your task should increase the size of the buffer and call RawKeyConvert again.

Inputs to Function

inputEvent A pointer to a task-defined buffer containing a series of InputEvent structures

bufferPointer A pointer to a task-defined buffer that will hold all ANSI byte values created by the conversion

bufferLength The number of bytes in the buffer

keyMap A pointer to a KeyMap structure that will convert raw key codes to ANSI bytes; if this value is null, the default KeyMap structure will be used

Preparation of the IOStdReq Structure

Initialize mn_ReplyPort to point to the MsgPort structure representing the desired task reply port. Initialize io_Device and io__Unit to point to the Device and ConUnit structures that manage unit -1 of the Console device. These parameters can always be copied from the IOStdReq structure initialized by an OpenDevice function call.

Discussion

The RawKeyConvert function uses a KeyMap structure to convert raw key codes to ANSI 3.64 bytes. The KeyMap structure can be either the KeyMap structure representing the current default key map or a KeyMap structure that is specified as part of the input definition of RawKeyConvert.

The ANSI bytes resulting from the conversion are placed into a task-defined buffer for further use by the task. You should always try to anticipate the maximum number of bytes for all conversions that your tasks will need to make. If the io_Length parameter value is large enough, the ANSI byte buffer will never overflow and the task can find

reliable bytes in it. Therefore, if RAM space is not at a premium, set io_Length large to guarantee the success of the RawKeyConvert function.

The RawKeyConvert and CDInputHandler functions both represent a direct entry into the Console device internal routines, which differs from the usual command-dispatching approach with BeginIO, DoIO, or SendIO.

STANDARD DEVICE COMMANDS

CMD_CLEAR

Purpose of Command

The CMD_CLEAR command clears the Console device read buffer, which is an internal device buffer used only for the CMD_READ command. Once the read buffer is cleared, subsequent CMD_READ commands can proceed from a known empty-buffer starting condition.

CMD_CLEAR allows QuickIO and only replies to the task reply-port queue if QuickIO is not successful. The results of command execution are found in io_Error, where 0 indicates that the command was successful. IOERR_NOCMD indicates that the io_Command parameter was specified incorrectly. IOERR_ABORTED indicates that the I/O request was aborted.

Preparation of the IOStdReq Structure

Initialize mn_ReplyPort to point to the MsgPort structure representing the desired task reply port. Initialize io_Device and io_Unit to point to the Device and ConUnit structures that manage each addressed unit of the Console device. These can always be copied from the IOStdReq structure initialized by the OpenDevice function call. Also initialize io_Command to CMD_CLEAR. Set io_Flags to 0, or set it to IOF_QUICK for QuickIO, which may or may not succeed depending on conditions in the system at the time CMD_CLEAR is dispatched.

Discussion

If a task is executing a series of CMD_READ commands and wants to ensure that the Console device internal read buffer is empty before it proceeds with those commands, it should first dispatch a CMD_CLEAR command to zero all bytes in the buffer and reset the buffer pointer. Then any CMD_READ commands subsequently dispatched by the task will not read extraneous characters left over from previous task operations.

CMD_CLEAR can be sent either as QuicklO or queued. If queued, it allows the task to execute any CMD_READ commands that are already queued before clearing the internal read buffer.

CMD_READ

Murpose of Command

The CMD_READ command causes one character or a stream of characters to be read into a task-defined buffer from the Amiga keyboard. These characters are buffered through the Console device unit internal read buffer. The input characters are in the form of an ANSI 3.64-byte stream; that is, they are either ASCII characters or escape-characters. Raw input events read by the Console device may be selectively filtered via the SRE (set raw events) and RRE (reset raw events) control sequences. Raw key codes are converted via the current Console device unit key map, which can be modified with the CD_ASKKEY-MAP and CD_SETKEYMAP commands.

CMD_READ allows QuicklO and only replies to the task reply-port queue if QuicklO is unsuccessful. The results of command execution are found in the io_Actual and io_Error parameters. The io_Actual parameter indicates the number of characters actually read. This will usually be 1 if a task specified the io_Length parameter as L A O value in io_Error indicates that the command was successful. IOERR_ABORTED indicates that the I/O request was aborted. IOERR_NOCMD indicates that the io_Command parameter was specified incorrectly, and IOERR_BADLENGTH indicates that the io_Length parameter was specified incorrectly.

Preparation of the IOStdReq Structure

Initialize mn_ReplyPort to point to the MsgPort structure representing the desired task reply port. Initialize io_Device and io_Unit to point to the desired Device and ConUnit structures that manage Console device unit 0. These can always be copied from the IOStdReq structure initialized by the OpenDevice function call. Set io_Command to CMD_READ. Also initialize the following command-specific parameters:

- io_Flags. Set this to 0 if not used; otherwise, set it to IOF_QUICK for QuicklO, which may or may not succeed depending on conditions in the system when CMD_READ is dispatched.

- io_Length. Set this to the number of characters to read from the Amiga keyboard. This is usually 1, indicating that the task is trying to read one character at a time. A different CMD_READ command will be dispatched for each character to be read.

- io_Data. Set this to point to the task's read (input) buffer, where the characters coming in from the keyboard through the Console device internal read buffer will eventually be placed.

Discussion

CMD_READ allows a task to place data into a task-defined buffer. The data usually comes from the Amiga keyboard; however, other hardware input devices can also use CMD_READ to place characters into a task-defined buffer for further processing. Each character read is part of an ANSI 3.64-byte stream consisting of ASCII characters or escape characters. A task can also request raw input events using the SRE (set raw events) and RRE (reset raw events) commands.

Usually the CMD_READ command is dispatched in order to request keyboard input one character at a time. However, if the IOStdReq structure io_Length parameter is specified as a value other than 1, CMD_READ will read multiple keyboard characters in succession, and the io_Actual parameter value will reflect the number actually read. It is therefore the responsibility of the task to look at the io_Actual parameter to verify how many characters the CMD_READ command actually returned.

Keyboard keys whose uppercase labels are ANSI standard characters (A, B, and so on) will usually be translated into their ASCII equivalent characters by the Console device internal routines using the current key map as defined by CD_ASKKEYMAP and CD_SETKEYMAP. For keys that do not have normal ASCII equivalents, an escape sequence is generated and inserted into the task's input stream. For example, in the default state, with no raw input events selected as RRE (reset raw events), the function keys Fl through F10 and the arrow keys will cause a set of escape sequences to be inserted into the input stream.

If a CMD_READ command is dispatched and the keyboard is not supplying any characters to satisfy it, the command is pending. In this case, it will be replied with the io_Actual parameter set to 0, indicating that no characters were read. If the keyboard is supplying fewer characters than the CMD_READ io_Length parameter specified, CMD_READ will be satisfied, but the output io_Actual and input io_Length parameters will not agree.

CMD_WRITE

Purpose of Command

The CMD_WRITE command causes a stream of characters to be written from a task-defined buffer one at a time into an Intuition window associated with a Console device

unit. The number of characters written is specified in the IOStdReq structure io_Length parameter. The characters can be displayed ASCII screen characters (hexadecimal 20 through 7E and A0 through FF) and screen control characters. The Intuition Window structure together with the Intuition internal routines determine and automatically control how many lines and how many characters per line can be displayed in the window, thus controlling word wrap in the window.

Most screen control characters (for example. Backspace and Return) are translated into their exact ANSI actions. The linefeed character is translated into a newline character. See the *ROM Kernel Manual* for other screen control characters.

If CMD_WRITE is specified as QuicklO but is unsuccessful, the request is treated as a queued I/O request and is replied to the calling task reply-port queue. The results of command execution are always found in the io_Error parameter, where 0 indicates that the command was successful. IOERR_ABORTED indicates that the I/O request was aborted. IOERR_NOCMD indicates that the io_Command parameter was specified incorrectly, and IOERR_BADLENGTH indicates that the io_Length parameter was specified incorrectly.

Preparation of the IOStdReq Structure

Initialize mn_ReplyPort to point to the MsgPort structure representing the desired task reply port. Initialize io_Device and io_Unit to point to the desired Device and ConUnit structures that manage Console device unit 0. These can always be copied from the IOStdReq structure initialized by the OpenDevice function call. Set io_Command to CMD_WRITE. Also initialize the following command-specific parameters:

- io_Flags. Set this to 0 if not used; otherwise, set it to IOF_QUICK for QuicklO, which may or may not succeed depending on conditions in the system when CMD_WRITE is dispatched.
- io_Length. Set this to the number of characters to be sent to the Console device unit window; count all ASCII characters and screen-control characters in the task-defined buffer.
- io_Data. Set this to point to the task's write buffer, which contains all the characters that will be sent to the Console device unit.

Discussfon

CMD_WRITE allows a task to send data from a task-defined buffer to an Intuition window associated with a Console device unit. This is how a task sends display characters to an Intuition window and controls the formatting of characters in that window.

The association between an Intuition window and a Console device unit is made by the OpenDevice function call. One CMD_WRITE command can be dispatched to any number of Console device units (Intuition windows) by changing the ConUnit cu_MP substructure name in the OpenDevice function call as required; see "Read-Write Operations" in this

chapter for more details. The RastPort structure associated with the Intuition Window structure is considered to be in use while the CMD_WRITE command is queued.

Screen control characters are used to format the ASCII display characters and to otherwise control the display of characters in the window. They divide into two broad classes, as follows:

- Those that carry single hexadecimal values. These include the linefeed character, the vertical tab character, and six others as defined in the *ROM Kernel Manual.*
- Those requiring the 9B CSI (control sequence introducer) lead-in character. These include cursor control commands; line control commands (delete line, insert line, and so on); page-setting commands (set page length, set line length, and so on); the SRE (set raw events) and RRE (reset raw events) commands; and the window-status request command, which tells the task about the current bounds (upper and lower row and column positions) of text in the Intuition window associated with a specific Console device unit.

Because the screen control characters are very detailed, it is best to study the CMD_WRITE command INCLUDE file presentation to determine their specific meanings.

DEVICE-SPECIFIC COMMANDS

CD_ ASKDEFAULTKEYMAP

Purpose of Command

CD_ASKDEFAULTKEYMAP fills a task-defined buffer with KeyMap structure parameters that define the default key map for a specified Console device unit. The KeyMap structure initializes the key map used by all Console device units when the Console device is opened; it is also the default unit key map used by RawKeyConvert when a null KeyMap pointer parameter is specified as input to that function. CD_ASKDEFAULTKEYMAP allows QuickIO and only replies to the task reply-port queue if QuickIO is unsuccessful.

The results of command execution are found in the io_Error parameter; a 0 value indicates that the command was successful. IOERR_ABORTED indicates that the I/O request was aborted. IOERR_NOCMD indicates that the io_Command parameter was specified incorrectly, and IOERR_B AD LENGTH indicates that the io_Length parameter was specified incorrectly.

In addition, the RAM data buffer at RAM location io_Data will contain the parameters of a KeyMap structure representing the default Console device unit key map.

Preparation of the IOStdReq Structure

Initialize mn_RepIyPort to point to the MsgPort structure representing the desired task reply port. Initialize io_Device and io__Unit to point to the Device and ConUnit structures that manage Console device unit 0. These can always be copied from the IOStdReq structure initialized by the OpenDevice function call. Set io_Command to CD_ASKDEFAULTKEYMAP Also initialize the following command-specific parameters:

- io_Flags. Set this to 0 if not used. Otherwise, set it to IOF_QUICK for QuicklO; QuicklO may or may not succeed, depending on conditions in the system when CD_ASKDEFAULTKEYMAP is dispatched.
- io_Data. Initialize this to point to a task-defined RAM data buffer that will contain the default key map when it is copied from the KeyMap structure.
- io_Length. Initialize this to the number of bytes in the KeyMap structure; a task can use the C language sizeof operator for this purpose.

Discussion

The CD_ASKDEFAULTKEYMAP command copies the values in a KeyMap structure representing the default Console device unit key map into a task-defined buffer. The values in this buffer can then be used by the current task. In addition, if a null value is used in the call to the RawKeyConvert function, the same KeyMap structure will be used to translate raw key code to ANSI key code; see the discussion of RawKeyConvert for more information.

CD_ASKKEYMAP

Purpose of Command

CD_ASKKEYMAP fills a task-defined buffer with the KeyMap structure parameters that define the current key map for a specified Console device unit. The KeyMap structure initializes the key map used by all Console device units when the Console device unit is opened. CD_ASKKEYMAP allows QuicklO and only replies to the task reply-port queue if QuicklO is unsuccessful.

The results of command execution are found in the io_Error parameter. A 0 value indicates that the command was successful. IOERR_ABORTED indicates that the I/O request was aborted. IOERR_NOCMD indicates that the io_Command parameter was incorrectly specified, and IOERR_BADLENGTH indicates that the io_Length parameter

was incorrectly specified. In addition, the RAM data buffer at RAM location io_Data contains the parameters of the KeyMap structure representing the current Console device unit key map.

Preparation of the IOStdReq Structure

Initialize mn_ReplyPort to point to the MsgPort structure representing the desired task reply port. Initialize io_Device and io_Unit to point to the desired Device and ConUnit structures that manage Console device unit 0. These can always be copied from the IOStdReq structure initialized by the OpenDevice function call. Set io_Command to CD_ASKKEYMAP. Also initialize the following command-specific parameters:

- io_Flags. Initialize this to IOF_QUICK for QuicklO; otherwise, set it to 0. QuicklO may or may not succeed, depending on conditions in the system when CD_ASKKEYMAP is dispatched.
- io_Data. Initialize this to point to a RAM data buffer that will contain the current unit key map after it is copied from the KeyMap structure.
- io_Length. Initialize this to the number of bytes in the KeyMap structure; a task can use the C language sizeof operator for this purpose.

Discussion

The CD_ASKDEFAULTKEYMAP and CD_ASKKEYMAP commands are similar; they allow a task to copy either the default or the current KeyMap structure parameters into a task-defined buffer. CD_ASKKEYMAP copies the values in a KeyMap structure representing the current—not necessarily the default—unit key map into a task-defined buffer. The values in this buffer can be used by the current task for its own specific needs. In contrast to the CD_ASKDEFAULTKEYMAP command, the CD_ASKKEY-MAP command does not supply a KeyMap structure for the RawKeyConvert function.

CD_SETDEFAULTKEYMAP

Purpose of Command

This command fills a KeyMap structure with data in a task-defined buffer containing KeyMap structure parameters. The KeyMap structure will then be used as the default key map for converting raw key code to ANSI 3.64 bytes, which are used for all Console device units and their associated Intuition windows. CD_SETDEFAULTKEYMAP allows QuicklO and only replies to the task reply-port queue if QuicklO is not successful.

The results of command execution are found in the io_Error parameter. A 0 indicates that the command was successful. IOERR_ABORTED indicates that the I/O request was aborted. IOERR_NOCMD indicates that the io_Command parameter was specified incorrectly, and IOERR_BADLENGTH indicates that the io_Length parameter was specified incorrectly. In addition, the current KeyMap structure will contain the KeyMap structure parameters copied from the task-defined buffer.

Preparation of the IOStdReq Structure

Initialize mn_ReplyPort to point to the MsgPort structure representing the desired task reply port. Initialize io_Device and io_Unit to point to the Device and ConUnit structures that manage Console device unit 0. These can always be copied from the IOStdReq structure initialized by the OpenDevice function call. Set io_Command to CD_SETDEFAULTKEYMAP. Also initialize the following command-specific parameters:

- io_Flags. Set this to 0 if not used, or set it to IOF_QUICK for QuicklO, which may or may not succeed depending on current conditions in the system.
- io_Data. Initialize this to point to a RAM data bufFer containing a KeyMap structure. The KeyMap structure at this location will then become the current key map used by the Console device for all Intuition windows attached to its units.
- io_Length. Initialize this to the number of bytes in the KeyMap structure] a task can use the C language sizeof operator for this purpose.

Discussion

The CD_SETDEFAULTKEYMAP command copies the values found in a task-defined buffer into the KeyMap structure representing the Console device unit default key map. The values in the key map can then be used by the current task for converting raw key codes to ANSI bytes.

CD_SETKEYMAP

Purpose of Command

CD_SETKEYMAP fills a KeyMap structure with data in a task-defined buffer that contains KeyMap structure parameters. This KeyMap structure will then be used as the current key map for the conversion of raw key codes to ANSI 3.64 bytes, which are used for all Console device units and their associated Intuition windows. CD_SETKEYMAP allows QuicklO and only replies to the task reply-port queue if QuicklO is not successful.

The results of command execution are found in io_Error; 0 indicates that the command was successful. IOERR_ABORTED indicates that the I/O request was aborted. IOERR_NOCMD indicates that the io_Command parameter was specified incorrectly, and IOERR_BADLENGTH indicates that the io_Length parameter was specified incorrectly. In addition, the current KeyMap structure will contain the KeyMap structure parameters copied from the task-defined buffer.

Preparation of the IOStdReq Structure

Initialize mn_ReplyPort to point to the MsgPort structure representing the desired task reply port. Initialize io_Device and io_Unit to point to the Device and ConUnit structures that manage Console device unit 0. These can always be copied from the IOStdReq structure initialized by the OpenDevice function call. Set io_Command to CD_SETKEYMAP. Also initialize the following command-specific parameters:

- io_Flags. Set this to 0 if not used; otherwise, set it to IOF_QUICK for QuicklO, which may or may not succeed depending on conditions in the system when CD_SETKEYMAP is dispatched.
- io_Data. Initialize this to point to a RAM data buffer containing a KeyMap structure. The KeyMap structure at this location will then become the current key map used by the Console device for all Intuition windows attached to its units.
- io_Length. Initialize this to the number of bytes in the KeyMap structure; a task can use the C language sizeof operator for this purpose.

Discussion

The CD_SETKEYMAP command copies the values found in a task-defined buffer into the KeyMap structure representing the current (not necessarily the default) Console device unit key map. The values in this key map can then be used by the current task to convert raw key codes to ANSI bytes. CD_SETKEYMAP does not supply a KeyMap structure for the RawKeyConvert function.