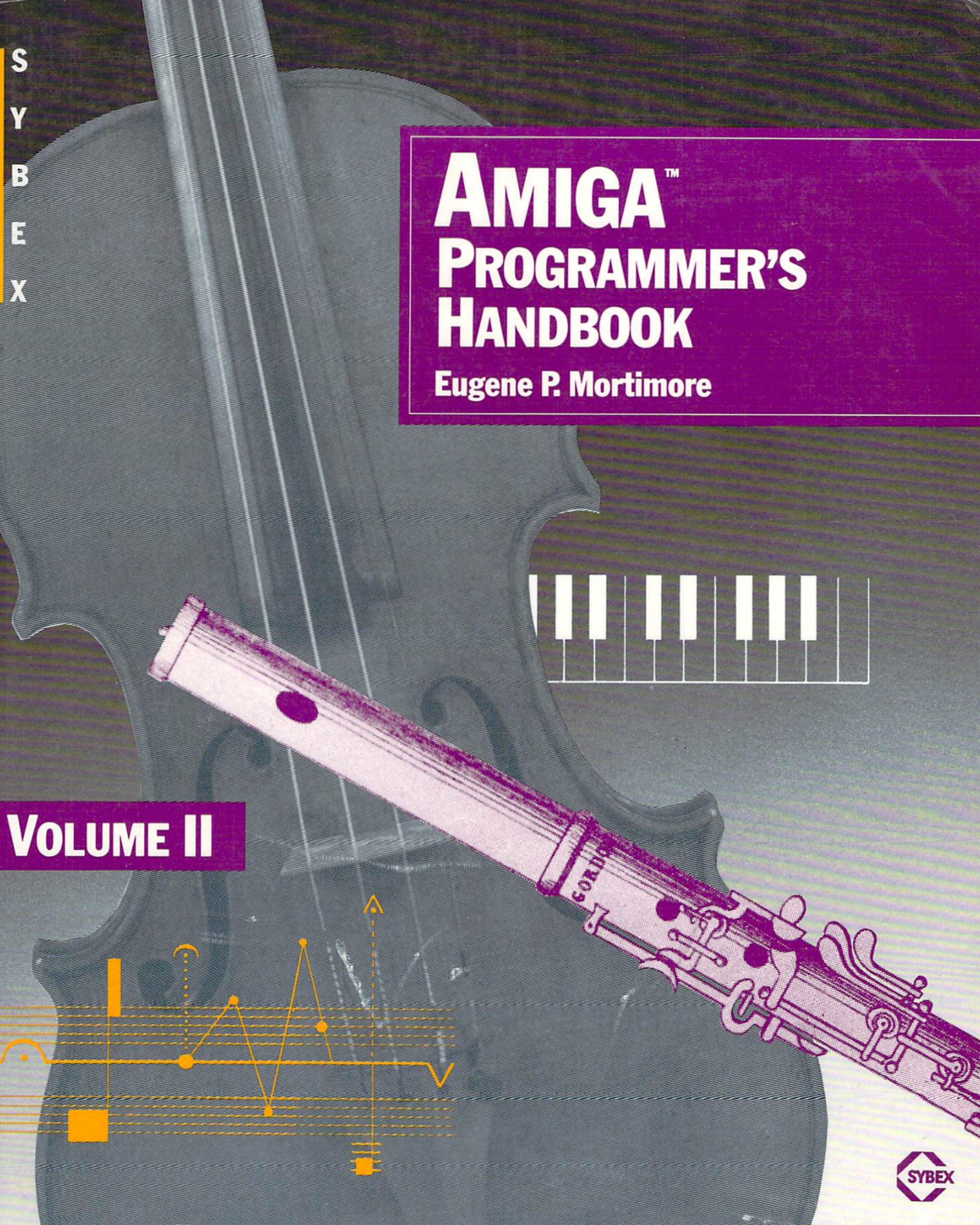
S
Y
B
E
X

# AMIGA™ PROGRAMMER'S HANDBOOK

Eugene P. Mortimore

VOLUME II



SYBEX

# VOLUME II

# AMIGA PROGRAMMER'S HANDBOOK

# AMIGA™ Programmer's Handbook

VOLUME II / SECOND EDITION

EUGENE P. MORTIMORE

SYBEX®

San Francisco ▪ Paris ▪ Düsseldorf ▪ London

Book design by Amparo Del Rio, Graphic Design

Illustrations by Jeffrey James Giese
Cover art by Thomas Ingalls + Associates

Library of Congress Card Number: 86-62805
ISBN 0-895888-384-8
Manufactured in the United States of America
10 9 8 7 6 5 4

Second edition Copyright© 1987 by SYBEX, Inc.

I *dedicate this book to a wonderful lady named Mary Alice.*

## ACKNOWLEDGMENTS

I would like to thank Neil Katin, developer of the Amiga Timer and TrackDisk devices, for his help in resolving technical questions. Robert Peck, author of the SYBEX *Programmer's Guide to the Amiga,* also provided useful device information.

At Commodore Busines Machines, Inc., Amiga Technical Support, I would like to thank Lisa A. Siracusa and Philip Lindsey for their help in obtaining up-to-date Amiga information and resolving technical questions.

I would also like to thank Rudolph Langer and Karl Ray for their helpful suggestions and guidance in the organization of this project.

I want to especially thank Valerie Robbins, SYBEX Developmental Editor, for her guidance in organizing, preparing, and editing the manuscript. It is always a pleasure to work with you, Valerie!

Also, Kay Luthin deserves special thanks for her attentive and patient editing and her help in organizing the book. Thanks Kay, you were a big help!

Special thanks also to Olivia Shinomoto for her expert word processing in preparing the manuscript. Thank you Olivia!

Finally, thanks to Gladys Varon, typesetter, and Jeff Green, proofreader.

# TABLE OF CONTENTS

## INTRODUCTION

xxi **Summary of Chapters**

xxiii **What the Amiga Can Do**

xxiv **The Device System**
- Programming Procedures xxiv
- Task–Device Sharing xxvii

xxvii **The Amiga Programming Environment**
- The Kickstart Disk xxviii
- The Internal Disk xxix
- The External Disk xxx

## 1 DEVICE I/O

1 **Introduction**

1 **Task-Device Interactions**

3 **Device I/O Request Classes**
- Queued I/O 3
- QuickIO 4

5 **Interactions with Multiple Ports and Units**
- Multiple Reply Ports 5
- Multiple Units 6

6 **Queue Behavior**
- Task Reply-Port Queue Behavior 6
- Device Message-Port Queue Behavior 8

8 **Shared Access Versus Exclusive Access**

10 **Multitasking and I/O Request Processing**

12 **Amiga Devices**

14 **Standard Device Commands**

16 **Device Functions**

16 **Structure Linkages for Tasks and Devices**

18 **General I/O Structures in the Amiga System**
- The IORequest Structure 18
- The IOStdReq Structure 20
- The Unit Structure 21
- The MsgPort Structure 21
- The Message Structure 22

23 **Device-Related Structures and INCLUDE Files**

## 2 DEVICE MANAGEMENT

27 **Introduction**

27 **General Programming Procedures**
- Asynchronous I/O Request Processing 30
- Synchronous I/O Request Processing 32

Multiple Tasks, Reply Ports, and Device Interaction 33
Immediate-Mode Request Processing 34

35 **General I/O Request-Structure Procedures**
Classes of I/O Requests 37
Creating Multiple I/O Requests 37
Processing Multiple I/O Requests 38

40 **Device Library Functions**
The AbortIO Function 41
The BeginIO Function 41

42 **The RemDevice and AddDevice Functions**

44 **Use of Exec-Support Library Functions**
CreateExtIO 44
CreatePort 45
CreateStdIO 46
CreateTask 47
DeleteExtIO 48
DeletePort 49
DeleteStdIO 50
DeleteTask 50
NewList 51

## 3 THE AUDIO DEVICE

55 **Introduction**

55 **Audio System Hardware**

57 **The Channel Combination Array**

59 **Channel Allocation Precedence**
Channel Locking and Stealing 59
Informing a Task of a Sound 60

60 **The Allocation Key**

61 **Audio Device Commands**
Sending Commands to the Audio Device 62

63 **Structures for the Audio Device**
The IOAudio Structure 63
The AudChannel Structure 66

66 **Audio Device Error Codes**

67 **Use of Functions**
CloseDevice 67
OpenDevice 68

70 **Standard Device Commands**
CMD_CLEAR 70
CMD_FLUSH 71
CMD_READ 72
CMD_RESET 73
CMD_START 74
CMD_STOP 75
CMD_UPDATE 76
CMD_WRITE 77

79 **Device-Specific Commands**
ADCMD_ALLOCATE 79
ADCMD_FINISH 81
ADCMD_FREE 82
ADCMD_LOCK 83
ADCMD_PERVOL 85
ADCMD_SETPREC 86
ADCMD_WAITCYCLE 87

# 4 THE NARRATOR DEVICE

91 **Introduction**

91 **English-String Processing**

92 **Narrator Device Commands**
- Sending Commands to the Narrator Device 92

94 **Structures for the Narrator Device**
- The Narrator_rb Structure 94
- The Mouth_rb Structure 96

97 **Narrator Device Error Codes**

98 **Use of Functions**
- CloseDevice 98
- CloseLibrary 99
- OpenDevice 99
- OpenLibrary 102
- Translate 103

104 **Standard Device Commands**
- CMD_FLUSH 104
- CMD_READ 105
- CMD_RESET 107
- CMD_START 107
- CMD_STOP 108
- CMD_WRITE 109

# 5 THE PARALLEL DEVICE

115 **Introduction**

115 **Read-Write Operations for the Parallel Device**

116 **Parallel Device Commands**
- Sending Commands to the Parallel Device 116

119 **Structures for the Parallel Device**
- The IOPArray Structure 119
- The IOExtPar Structure 120
- Parallel-Port Flag Parameters 120

121 **Use of Functions**
- CloseDevice 121
- OpenDevice 122

124 **Standard Device Commands**
- CMD_FLUSH 124
- CMD_READ 125
- CMD_RESET 127
- CMD_START 128
- CMD_STOP 129
- CMD_WRITE 130

131 **Device-Specific Commands**
- PDCMD_QUERY 131
- PDCMD_SETPARAMS 133

# 6 THE SERIAL DEVICE

137 **Introduction**

137 **Read-Write Operations for the Serial Device**

139 **Serial Device Commands**
- Sending Commands to the Serial Device 139

140 **Structures for the Serial Device**
- The IOTArray Structure 140
- The IOExtSer Structure 142
- Serial-Port Flag Parameters 143

146 **Use of Functions**
CloseDevice 146
OpenDevice 147

149 **Standard Device Commands**
CMD_CLEAR 149
CMD_FLUSH 149
CMD_READ 150
CMD_RESET 153
CMD_START 154
CMD_STOP 155
CMD_WRITE 156

158 **Device-Specific Commands**
SDCMD_BREAK 158
SDCMD_QUERY 160
SDCMD_SETPARAMS 161

# 7 THE INPUT DEVICE

167 **Introduction**

168 **Operation of the Input Device**

168 **Using Input-Handler Functions**

171 **Input Device Commands**
Sending Commands to the Input Device 171

172 **Structures for the Input Device**
The InputEvent Structure 173

175 **Use of Functions**
CloseDevice 175
OpenDevice 176

178 **Standard Device Commands**
CMD_FLUSH 178
CMD_RESET 178
CMD_START 179
CMD_STOP 180

181 **Device-Specific Commands**
IND_ADDHANDLER 181
IND_REMHANDLER 182
IND_SETMPORT 183
IND_SETMTRIG 184
IND_SETMTYPE 184
IND_SETPERIOD 186
IND_SETTHRESH 187
IND_WRITEEVENT 188

# 8 THE CONSOLE DEVICE

193 **Introduction**

193 **Operation of the Console Device**
Read–Write Operations for the Console Device 193

197 **Console Device Commands**
Sending Commands to the Console Device 197

198 **Structures for the Console Device**
The ConUnit Structure 199
The KeyMap Structure 202
The KeyMapNode Structure 203
The KeyMapResource Structure 204

204 **Use of Functions**
CDInputHandler 204
CloseDevice 205
OpenDevice 206
RawKeyConvert 208

210 **Standard Device Commands**
- CMD_CLEAR 210
- CMD_READ 211
- CMD_WRITE 212

214 **Device-Specific Commands**
- CD_ASKDEFAULTKEYMAP 214
- CD_ASKKEYMAP 215
- CD_SETDEFAULTKEYMAP 216
- CD_SETKEYMAP 217

# 9 THE KEYBOARD DEVICE

221 **Introduction**

221 **Operation of the Keyboard Device**
- Keyboard Input Event Processing 222

224 **Keyboard Device Commands**
- Sending Commands to the Keyboard Device 224

225 **Structures for the Keyboard Device**

225 **Use of Functions**
- CloseDevice 225
- OpenDevice 226

228 **Standard Device Commands**
- CMD_CLEAR 228
- CMD_RESET 229

230 **Device-Specific Commands**
- KBD_ADDRESETHANDLER 230
- KBD_READEVENT 231
- KBD_READMATRIX 233
- KBD_REMRESETHANDLER 235
- KBD_RESETHANDLERDONE 236

# 10 THE GAMEPORT DEVICE

239 **Introduction**

240 **Operation of the Gameport Device**
- Gameport Input Event Processing 242

243 **Gameport Device Commands**
- Sending Commands to the Gameport Device 243

244 **Structures for the Gameport Device**
- The GamePortTrigger Structure 244

245 **Use of Functions**
- CloseDevice 245
- OpenDevice 246

248 **Standard Device Commands**
- CMD_CLEAR 248

249 **Device-Specific Commands**
- GPD_ASKCTYPE 249
- GPD_ASKTRIGGER 250
- GPD_READEVENT 251
- GPD_SETCTYPE 253
- GPD_SETTRIGGER 255

# 11 THE PRINTER DEVICE

259 **Introduction**

259 **Operation of the Printer Device**
- Sending Control Codes to a Printer 261

263 **Printer Device Commands**
- Sending Commands to the Printer Device 263

265 **The PrinterIO Union**

265 **Structures for the Printer Device**
- The IOPrtCmdReq Structure 266
- The IODRPReq Structure 267

268 **Use of Functions**
- CloseDevice 268
- OpenDevice 269

271 **Standard Device Commands**
- CMD_FLUSH 271
- CMD_RESET 272
- CMD_START 272
- CMD_STOP 273
- CMD_WRITE 274

275 **Device-Specific Commands**
- PRD_DUMPRPORT 275
- PRD_PRTCOMMAND 278
- PRD_RAWWRITE 279

# 12 THE CLIPBOARD DEVICE

283 **Introduction**

283 **Operation of the Clipboard Device**
- Ordinal Clip Identifiers 284
- Sequential Read–Write Operations for a Clipboard File 284

286 **Clipboard Device Commands**
- Sending Commands to the Clipboard Device 286

287 **Structures for the Clipboard Device**
- The ClipboardUnitPartial Structure 288
- The IOClipReq Structure 288
- The SatisfyMsg Structure 290

290 **Use of Functions**
- CloseDevice 290
- OpenDevice 291

293 **Standard Device Commands**
- CMD_READ 293
- CMD_RESET 295
- CMD_UPDATE 295
- CMD_WRITE 296

298 **Device-Specific Commands**
- CBD_CLIPREADID 298
- CBD_CLIPWRITEID 299
- CBD_POST 300

# 13 THE TIMER DEVICE

305 **Introduction**

305 **Operation of the Timer Device**

306 **Timer Units and Timer Arithmetic**
- Time-Interval Corrections in a Busy System 307

309 **Timer Device Commands**
- Sending Commands to the Timer Device 309

310 **Structures for the Timer Device**
- The TimeVal Structure 310
- The TimeRequest Structure 311

311 **Use of Functions**
AddTime 311
CloseDevice 312
CmpTime 313
OpenDevice 314
SubTime 316

317 **Device-Specific Commands**
TR_ADDREQUEST 317
TR_GETSYSTIME 318
TR_SETSYSTIME 319

**14 THE TRACKDISK DEVICE**

323 **Introduction**

323 **Operation of the TrackDisk Device**
TrackDisk Device/Floppy Disk Interactions 325

326 **TrackDisk Device Commands**
Sending Commands to the TrackDisk Device 326

327 **Structures for the TrackDisk Device**
The IOExtTD Structure 328
The TDUPublicUnit Structure 329
The BootBlock Structure 330

330 **TrackDisk Device Error Codes**

332 **Use of Functions**
CloseDevice 332
OpenDevice 332

333 **Device-Specific Commands**
ETD_CLEAR 333
ETD_RAWREAD 334
ETD_RAWWRITE 335
ETD_UPDATE 336
TD_ADDCHANGEINT 337
TD_CHANGENUM 338
TD_CHANGESTATE 338
TD_FORMAT 339
TD_GETDRIVETYPE 340
TD_GETNUMTRACKS 340
TD_MOTOR 341
TD_PROTSTATUS 342
TD_RAWREAD 342
TD_RAWWRITE 344
TD_REMCHANGEINT 345
TD_REMOVE 345
TD_SEEK 346

**APPENDIX**

C Language Definitions of the Exec-Support Library Functions

351 **CreateExtIO**

352 **CreatePort**

352 **CreateStdIO**

353 **CreateTask**

354 **DeleteExtIO**

354 **DeletePort**

355 **DeleteStdIO**

355 **DeleteTask**

**Index**

# Introduction

# Introduction

This book is Volume II of a two-volume series. Volume I presented the graphics-related Amiga library functions. Volume II presents the 12 built-in Amiga devices. The two volumes provide a complete reference guide for a programmer who wants to use the Amiga graphics and device management features. For this reason, it is recommended that the reader also secure a copy of Volume I.

The material presented here is applicable to all three currently available Amiga machines: the original Amiga 1000, the new Amiga 500, and the new, highly expandable, IBM-compatible Amiga 2000. These machines are entirely software compatible, because they all use Release 1.2 software. Any Release 1.2 function or command documented in either volume operates identically for these three machines; properly written programs designed for one machine will always run on the other two machines as well. The three machines differ only in how some of the software libraries are loaded. Whereas the Amiga 1000 machine uses the Kickstart disk to load most built-in libraries, devices, and other supporting software, the 500 and 2000 machines have the fully debugged Release 1.2 software built directly into ROM.

Amiga devices are preprogrammed in a highly efficient manner not common to other microcomputers. The internal routines of each device are fully debugged and predefined; in turn, they define the functions and commands specific to each device. However, because these internal routines are predefined and therefore fixed, a programmer must be familiar with how they work in order to interface a program with them. Understanding task–device interactions is of critical importance for programming.

The C and assembly language INCLUDE files contain the information required for proper program interfacing. The greatest difficulty in device programming, therefore, is learning the rules and implementing them in your programs. In order to do this, you must understand the concepts of Amiga device programming. A set of figures presented in the first two chapters of this book illustrates the most important concepts of device programming. You will not find these diagrams in any other documentation; they provide the conceptual framework by which you can both understand and program Amiga devices.

Once you understand device-programming procedures, you must learn the meaning of all of the predefined structures, structure parameters, device-related parameters and flag parameter bits. You can then produce your own device programs in the most creative and efficient manner, using the predefined device routines and functions to maximum advantage.

## Summary of Chapters

Like Volume I, this book is written in reference format. The first two chapters present general device-programming concepts that you must understand in order to work with the devices successfully. The next 12 chapters discuss each of the 12 built-in devices individually. Each chapter begins with an overview of the device's requirements and behavior, illustrated by a comprehensive set of diagrams. The structures, functions, and commands used with the device are then examined in detail.

Chapter 1 discusses device I/O, including QuickIO and queued I/O, asynchronous and synchronous I/O requests, signals from devices to tasks and from tasks to devices, multiple tasks and units, the management of queues, and shared and exclusive access modes. The chapter also summarizes the structures, INCLUDE files, functions, and Amiga device libraries. Here we present the task–device message-port model that most devices follow. This model shows how a Unit structure is used to manage each open device unit.

Chapter 2 discusses device management, including general programming and I/O request-structure procedures. The use of the AbortIO and BeginIO functions and automatic immediate-mode commands is presented here, as well as the Exec-support library functions. These nine functions make the management of tasks, message ports, I/O request structures, and device-related queue lists much easier. (The programming statements for these functions are presented in the appendix.)

Chapter 3 discusses the Audio device, which allows a task to produce sound from stereo speakers. Because of the Amiga's multitasking capabilities and the presence of multiple audio channels, this device is the most complicated in the Amiga device system—both to understand and to program.

Chapter 4 discusses the Narrator device, which allows a task to make the Amiga "talk" by using synthetic speech algorithms. Synthesized speech requires the use of the Translator library Translate function as well as the Audio device; the chapter discusses this interaction.

Chapter 5 discusses the Parallel device, which allows a task to control the information passing back and forth between a set of task-defined buffers and external hardware attached to the parallel port. Chapter 6 provides the same kind of information for the Serial device.

Chapter 7 discusses the Input device, which allows a task to control the Amiga input event system. The Input device acts as a merging mechanism for input events coming from the keyboard, the gameports, the Timer device, and the disk system. A task can also add its own input events to this input event stream.

Chapter 8 discusses the Console device, which is responsible for processing input events coming from the keyboard through the Keyboard device internal routines and for sending formatted text information to Intuition windows. This device interacts with the Console, Input, TrackDisk, Keyboard, Timer, and Gameport devices. It does not precisely follow the task–device message-port model presented in Chapter 1; instead, it uses the ConUnit structure.

Chapter 9 discusses the Keyboard device, which allows a task to control the keyboard hardware system. The Keyboard device is responsible for creating input events for user-generated keyboard signals and for adding keyboard reset functions to the system.

Chapter 10 discusses the Gameport device, which allows a task to control the gameport hardware system. The Gameport device is responsible for creating input events from gameport signals generated by a mouse or another type of controller.

Chapter 11 discusses the Printer device, which allows a task to control the printer hardware. The Printer device is responsible for sending text and graphics to a printer connected to either the serial or parallel port.

Chapter 12 discusses the Clipboard device, which allows a task to control cut-and-paste operations. The Clipboard device manages the transfer of text characters between 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 preproduced 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 opened, 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 mode devices, the same Device and Unit structures are shared by any tasks that have the device's unit open. The Device structure lib_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 unit, 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 conditions. 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 memory 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 its 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 the 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.1 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, Printer, 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

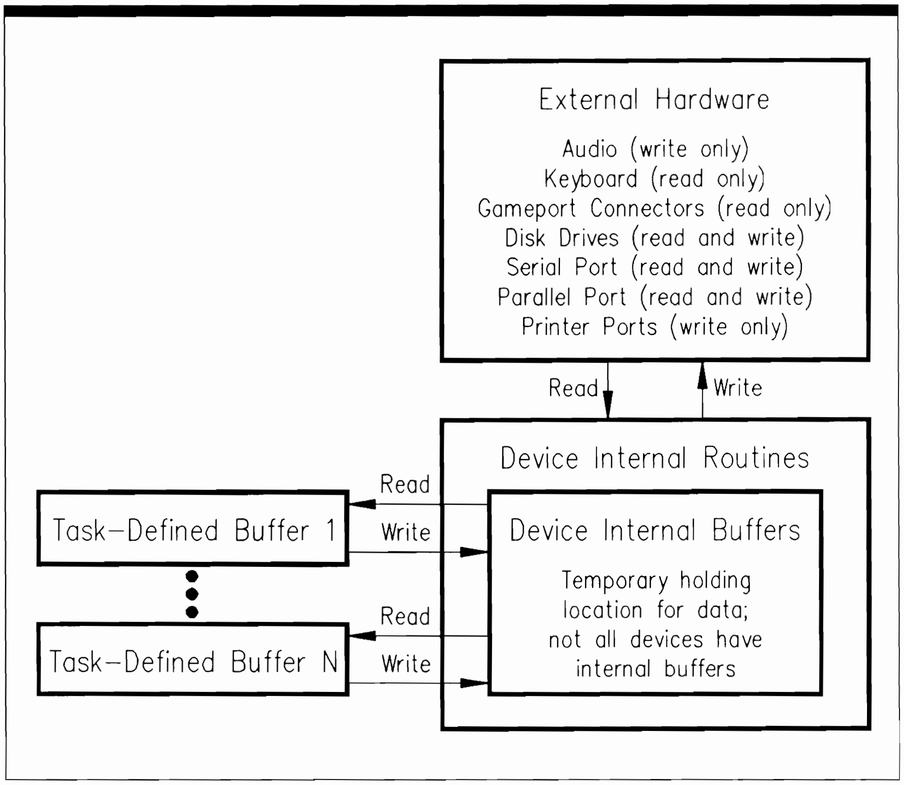

External Hardware
Audio (write only)
Keyboard (read only)
Gameport Connectors (read only)
Disk Drives (read and write)
Serial Port (read and write)
Parallel Port (read and write)
Printer Ports (write only)
Read
Write
Device Internal Routines
Read
Task-Defined Buffer 1
Write
Device Internal Buffers
Temporary holding location for data; not all devices have internal buffers
Read
Task-Defined Buffer N
Write

**Figure I.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 I.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.

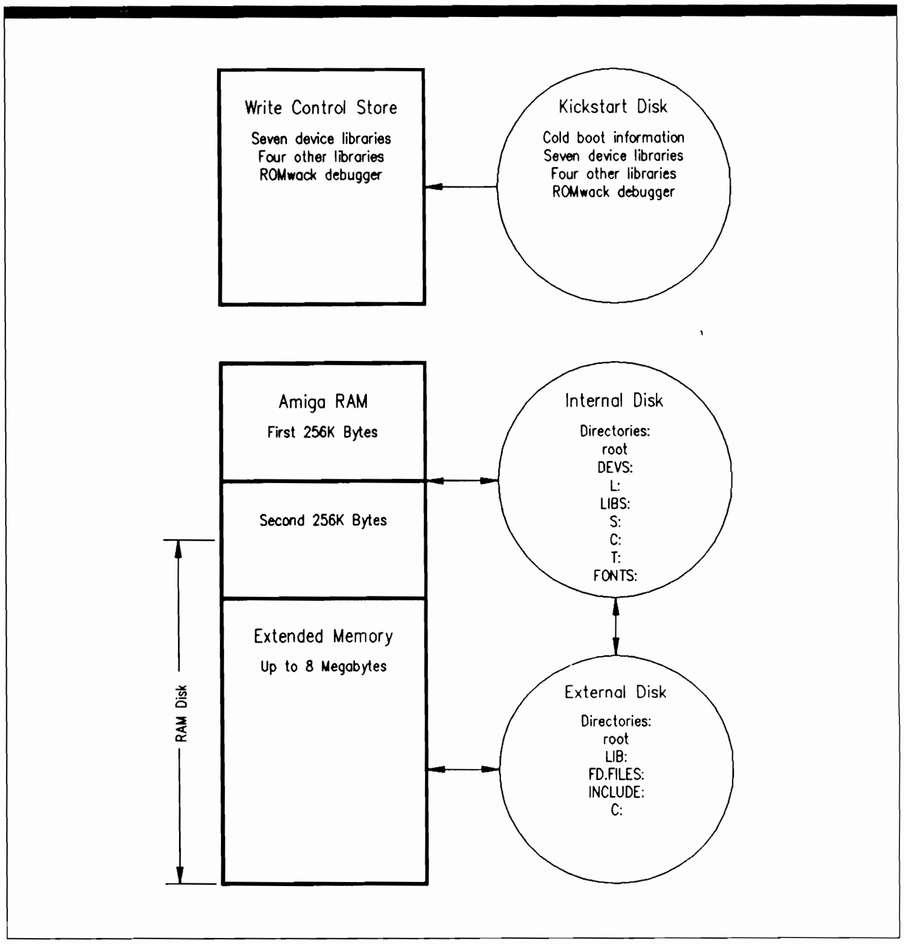


**Figure I.2:** *Amiga Programming Environment*

Total Amiga RAM consists of at least 512K of internal RAM and up to 8 megabytes of external RAM. The first 512K of internal RAM can be used as chip memory (MEMF-_CHIP, see Volume I), and the external RAM can be fast RAM (MEMF_FAST). 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 directory contains all AmigaDOS operating-system commands, represented as compiled, executable files. These include the DIR command, the DiskCopy 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 directory 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: directory. This directory contains the C language compiler and linking programs. For the Lattice compiler, these are called LC1, LC2, and Alink. LC1 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 object 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: directory. This directory must contain object files and the information needed to support a compiler's first- and second-pass programs (LC1 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 (LC1, 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.

# 1 Device I/O

## Introduction

This chapter discusses the general aspects of device I/O and task–device interactions. It presents important concepts 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

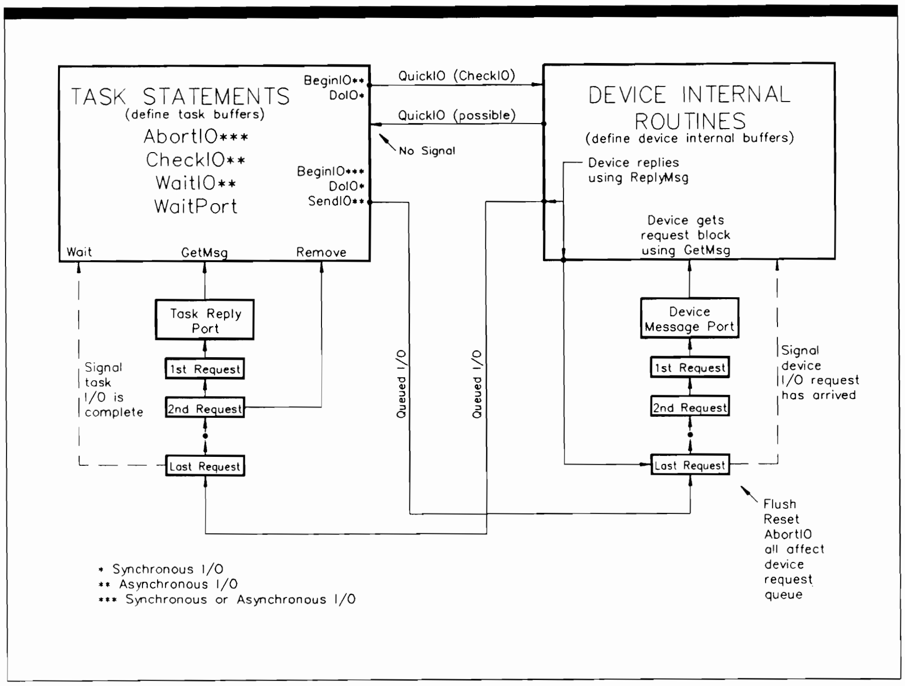


**Figure 1.1:** *Task–Device Interaction Using I/O Request Structures*

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, CloseDevice, BeginIO, DoIO, SendIO, AbortIO, CheckIO, 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 all 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 IO 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 IO 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 WaitPort function allows each task to wait for the reply of an I/O request. See Volume I for the distinctions between these two methods.

## QuickIO

The second major class of I/O is quick I/O, which will be referred to throughout this volume as QuickIO. With successful QuickIO, 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 QuickIO, 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 QuickIO is possible based on current conditions in the system. If the device is not currently busy, QuickIO 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.

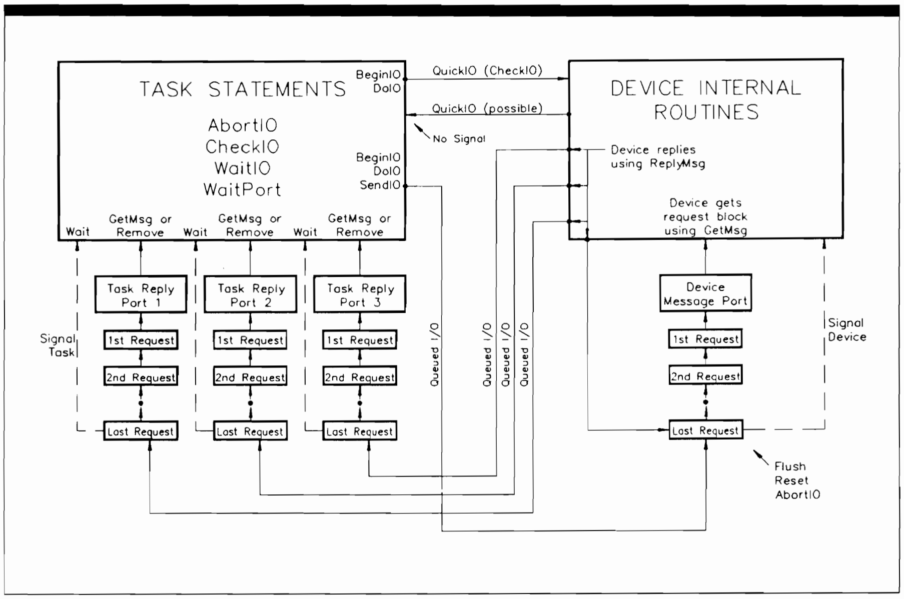


**Figure 1.2:** *Task-Device Interaction with Three Reply Ports*

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

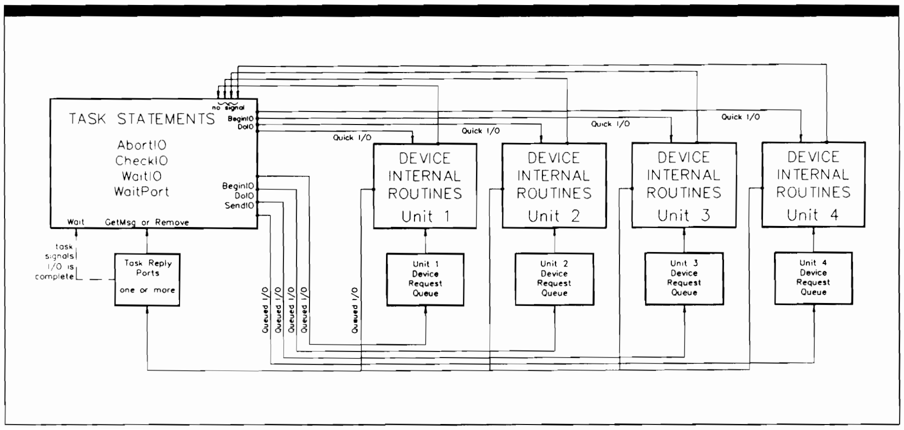
Figure 1.3 shows one task working with multiple units of a single device. Each device unit could be replying to one or more task reply ports. This is a useful configuration if, for example, you are working with all four units of the TrackDisk device and you want to place data from each disk drive in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



TASK STATEMENTS
no signal
BeginIO
DoIO
AbortIO
CheckIO
WaitIO
WaitPort
BeginIO
DoIO
SendIO
Wait
GetMsg or Remove
task signals I/O is complete
Task Reply Ports
one or more
Queued I/O
Queued I/O
Queued I/O
Queued I/O
Queued I/O
Quick I/O
Quick I/O
Quick I/O
Quick I/O
DEVICE INTERNAL ROUTINES Unit 1
DEVICE INTERNAL ROUTINES Unit 2
DEVICE INTERNAL ROUTINES Unit 3
DEVICE INTERNAL ROUTINES Unit 4
Unit 1 Device Request Queue
Unit 2 Device Request Queue
Unit 3 Device Request Queue
Unit 4 Device Request Queue

**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.

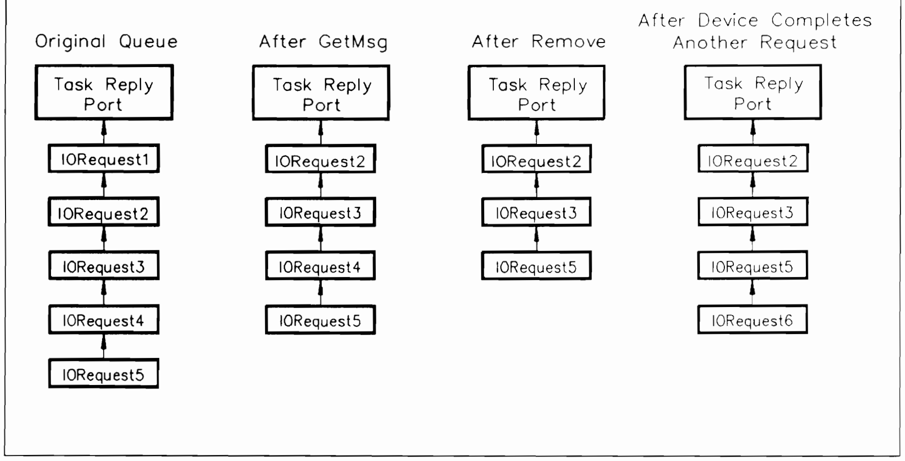


**Figure 1.4:** *Behavior of a Task's I/O Reply-Port Queue*

### 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. IORequest1 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

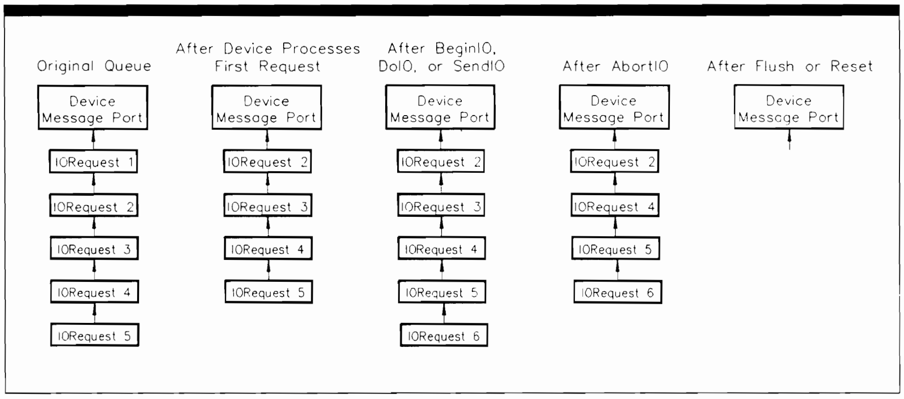


**Figure 1.5:** *Behavior of a Device's I/O Message-Port Queue*

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 Task1 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 Task1 must finish using the device before Task2 and Task3 can gain access to it. All of the BeginIO, DoIO, and SendIO functions in Task1 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 Task1 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 Task1 regains the CPU, executes a CloseDevice

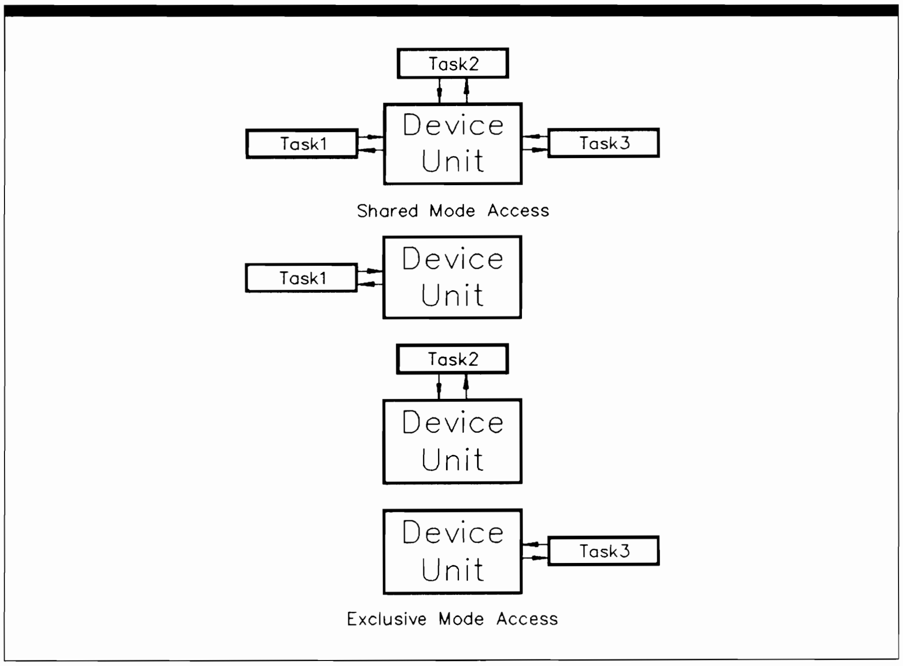


**Figure 1.6:** *Difference between Shared and Exclusive Access*

call, and closes that unit of the device. Once Task1 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 all 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 (Task1, 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. Task1 needs to send three I/O requests to the device unit: IORequest11, IORequest12, and IORequest13, 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 IORequest21, IORequest22, and IORequest23 to the device unit. Task3 needs to send IORequest31, IORequest32, and IORequest33.

In this example, Task1 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. Task1 issues a DoIO call to send IORequest11. 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 IORequest11 cannot be processed immediately and DoIO cannot return in Task1, Task1 will be blocked. The next-higher priority task will take over; by assumption, this is Task2.

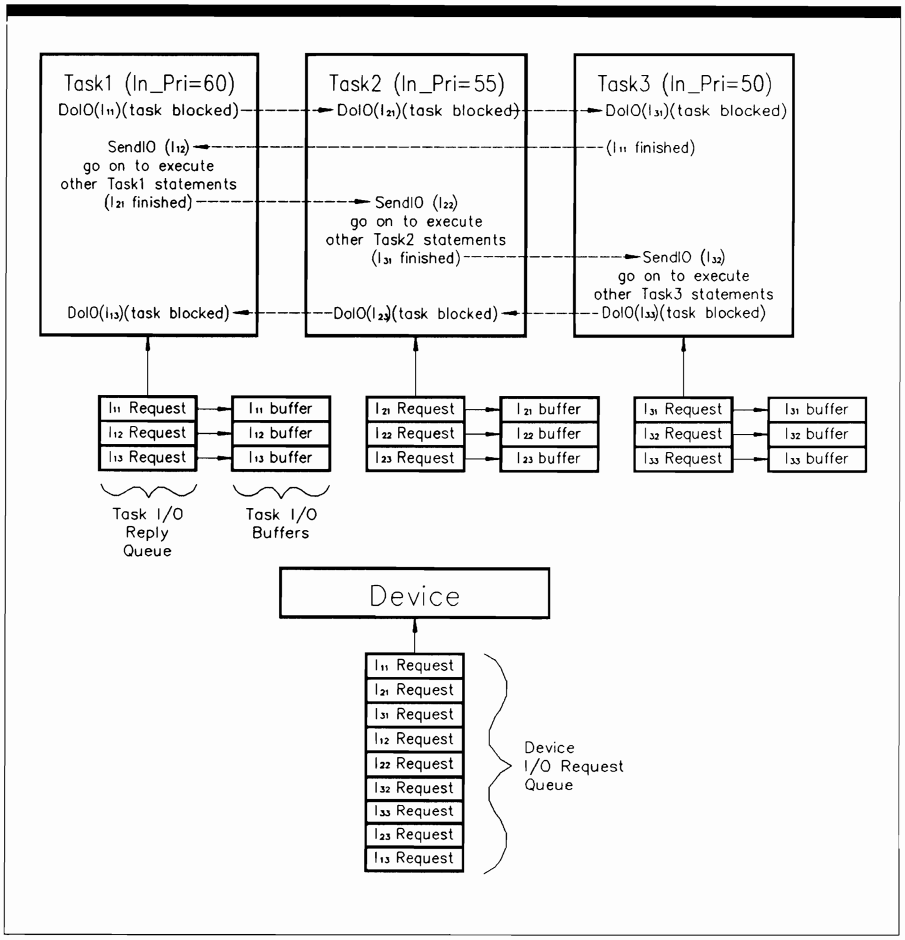


**Figure 1.7:** *Multitasking and I/O Request Processing*

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 IORequest11 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 IORequest11, IORequest21, and other I/O requests already in the queue. Task3 will now be blocked.

Now assume that the device internal routines just finished with IORequest11. (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 IORequest11 is processed.) The previously blocked Task1 can then get control of the CPU; Task1 has been waiting on IORequest11, and the device internal routines have signaled that IORequest11 is completed. Task1 receives the IORequest11 request in its task reply-port queue and acts on the arrival signal. This gives CPU control to Task1, indicated by the dotted line between the Task1 and Task3 rectangles. The device internal routines will signal Task1 (using the equivalent of the PutMsg function's signal mechanism) and Task1 can now go on to execute other task statements.

Now assume that the next task statement in Task1 is a SendIO function call. This call sends IORequest12 (asynchronous) to the queue, behind IORequest21, IORequest31, and other previously queued I/O requests. (Note that IORequest11 is no longer in the device-unit request queue).

Because SendIO sends an asynchronous I/O request, Task1 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 Task1 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, CheckIO, 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

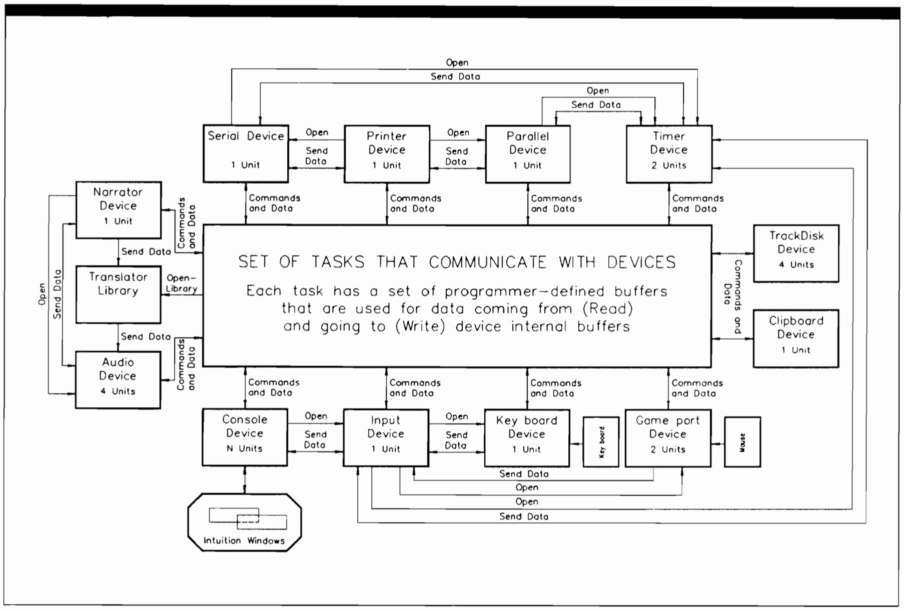


**Figure 1.8:** *Task–Device Relationships for All Amiga Devices*

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 eight 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 all 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_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).

| Device | COMMAND | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | CLEAR | FLUSH | READ | RESET | START | STOP | UPDATE | WRITE |
| Audio | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Clipboard | — | — | ✓ | ✓ | — | — | ✓ | ✓ |
| Console | ✓ | — | ✓ | — | — | — | — | ✓ |
| Gameport | ✓ | — | — | — | — | — | — | — |
| Input | — | ✓ | — | ✓ | ✓ | — | — | — |
| Keyboard | ✓ | — | — | ✓ | — | — | — | — |
| Narrator | — | ✓ | ✓ | ✓ | ✓ | ✓ | — | ✓ |
| Parallel | — | ✓ | ✓ | ✓ | ✓ | ✓ | — | ✓ |
| Printer | — | ✓ | — | ✓ | ✓ | ✓ | — | ✓ |
| Serial | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | — | ✓ |
| Timer | — | — | — | — | — | — | — | — |
| TrackDisk | — | — | ✓ | — | — | — | — | — |

**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.

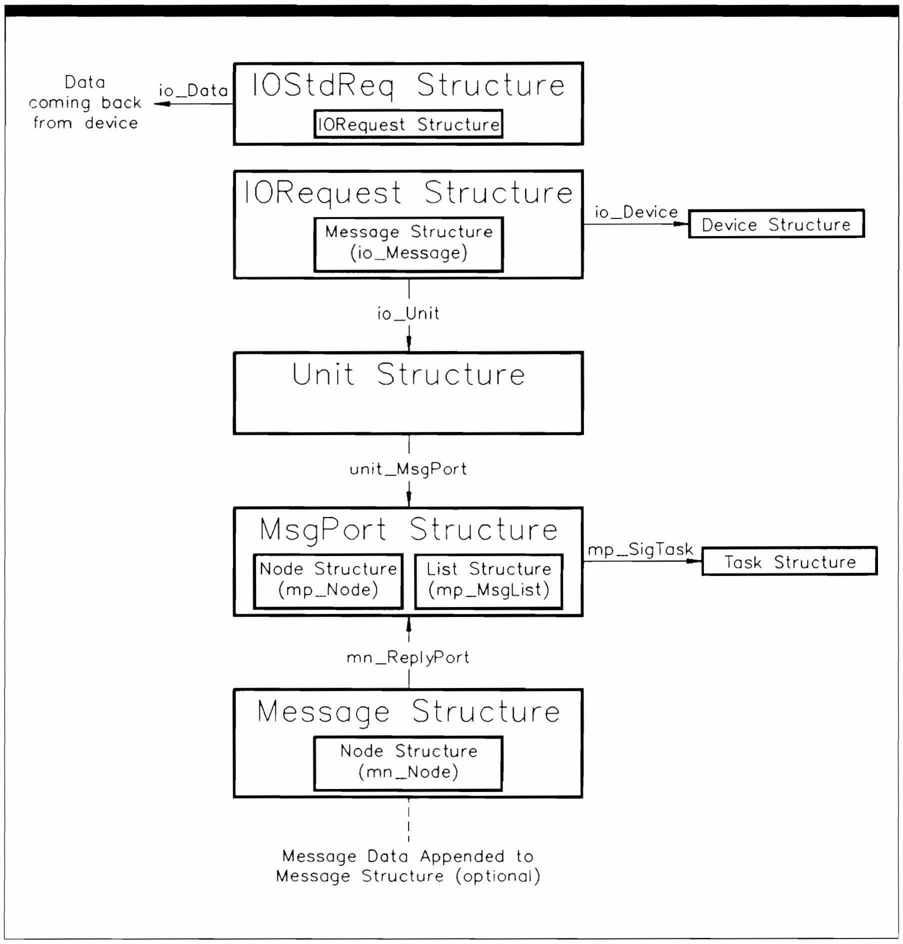


**Figure 1.9:** *Structure Linkages for General Task-Device I/O Processing*

The IOStdReq structure contains the IORequest 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 IORequest 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 IORequest structure also contains the io_Unit pointer, which points to the Unit structure used to manage the operation of one device unit. Just like the io_Device parameter, the io_Unit parameter is specified by the system when the OpenDevice function call returns.

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 QuickIO. Then the device will process the I/O request immediately if possible. If the device cannot handle the request as a QuickIO 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_Reserved1 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 ln_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.

| 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 | IOStdReq | — | 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 | IOPArray | — | — | — | Parallel.h |
| Printer | IOPrtCmdReq IODRPReq | IOStdReq | PrinterData | PrinterSegment | PrinterEx-tendedData | DeviceData | Prtbase.h Printer.h |
| Serial | IOExtSer | IOStdReq | IOTArray | — | — | — | Serial.h |
| Timer | TimeRequest | IORequest | TimeVal | — | — | — | Timer.h |
| TrackDisk | IOExtTD | IOStdReq | TDV_Public-Unit | — | — | — | Trackdisk.h |

**Table 1.2:** *Device Structures and INCLUDE Files*

# 2 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.

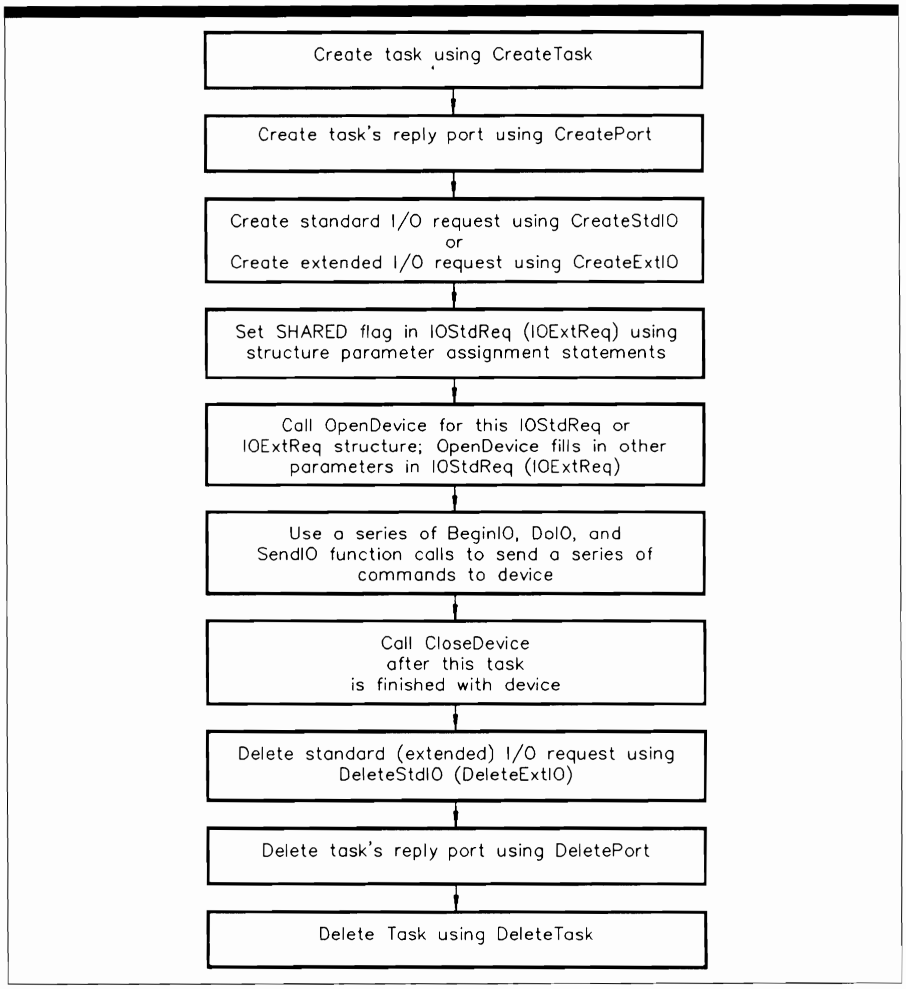

Create task using CreateTask
Create task's reply port using CreatePort
Create standard I/O request using CreateStdIO or Create extended I/O request using CreateExtIO
Set SHARED flag in IOStdReq (IOExtReq) using structure parameter assignment statements
Call OpenDevice for this IOStdReq or IOExtReq structure; OpenDevice fills in other parameters in IOStdReq (IOExtReq)
Use a series of BeginIO, DoIO, and SendIO function calls to send a series of commands to device
Call CloseDevice after this task is finished with device
Delete standard (extended) I/O request using DeleteStdIO (DeleteExtIO)
Delete task's reply port using DeletePort
Delete Task using DeleteTask

**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, SendIO, AbortIO, CheckIO, 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, QuickIO was not requested.

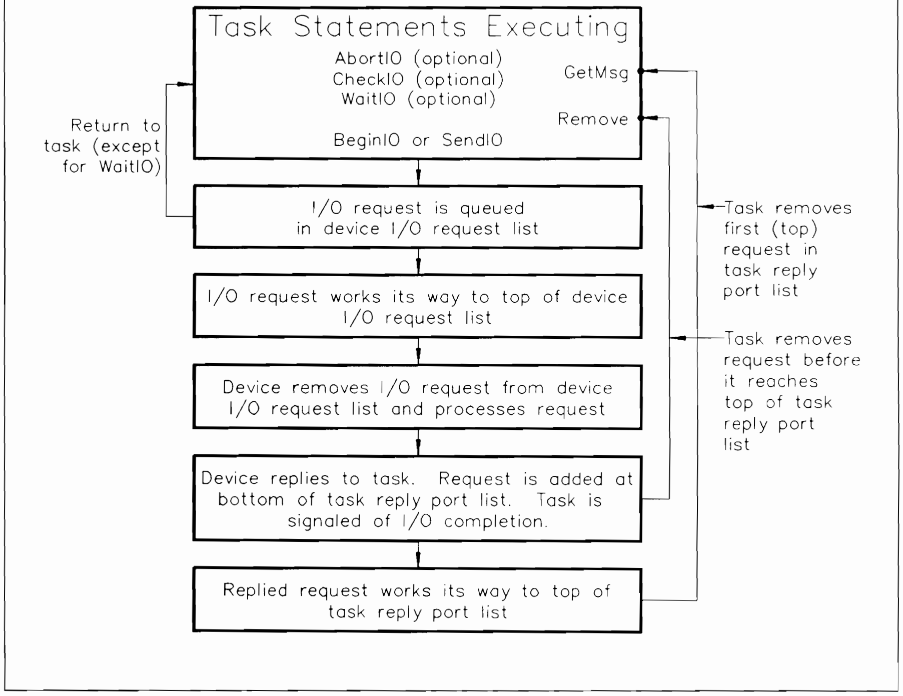

Task Statements Executing
AbortIO (optional)
CheckIO (optional)
WaitIO (optional)
GetMsg
Remove
BeginIO or SendIO
Return to task (except for WaitIO)
I/O request is queued in device I/O request list
I/O request works its way to top of device I/O request list
Device removes I/O request from device I/O request list and processes request
Device replies to task. Request is added at bottom of task reply port list. Task is signaled of I/O completion.
Replied request works its way to top of task reply port list
Task removes first (top) request in task reply port list
Task removes request before it reaches top of task reply port list

**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.

## 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

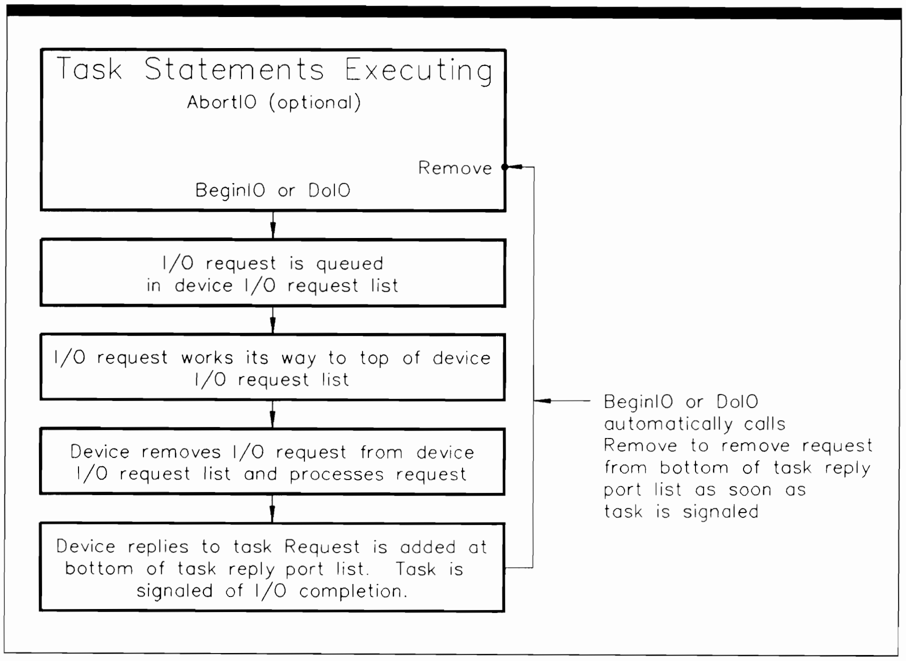

Task Statements Executing
AbortIO (optional)
Remove
BeginIO or DoIO
I/O request is queued in device I/O request list
I/O request works its way to top of device I/O request list
Device removes I/O request from device I/O request list and processes request
Device replies to task Request is added at bottom of task reply port list. Task is signaled of I/O completion.
BeginIO or DoIO automatically calls Remove to remove request from bottom of task reply port list as soon as task is signaled

**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 Task1. Task3, in turn, was created by a CreateTask call in Task2. Thus, Task3 also was created indirectly by Task1.

As an example of how this works, think of Device1 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: Task1 needs data from Device1 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.

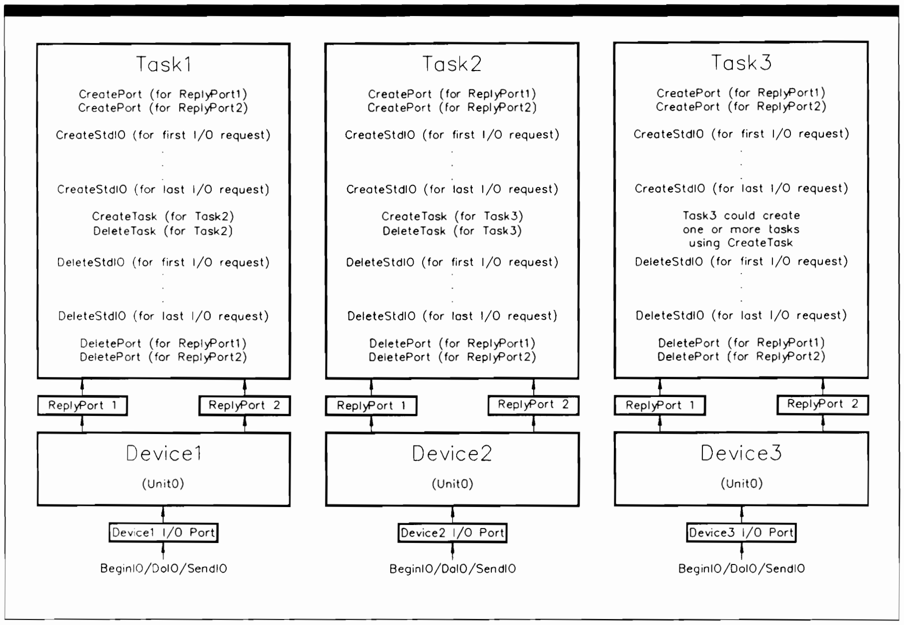

Task1
CreatePort (for ReplyPort1)
CreatePort (for ReplyPort2)
CreateStdIO (for first I/O request)
CreateStdIO (for last I/O request)
CreateTask (for Task2)
DeleteTask (for Task2)
DeleteStdIO (for first I/O request)
DeleteStdIO (for last I/O request)
DeletePort (for ReplyPort1)
DeletePort (for ReplyPort2)
Task2
CreatePort (for ReplyPort1)
CreatePort (for ReplyPort2)
CreateStdIO (for first I/O request)
CreateStdIO (for last I/O request)
CreateTask (for Task3)
DeleteTask (for Task3)
DeleteStdIO (for first I/O request)
DeleteStdIO (for last I/O request)
DeletePort (for ReplyPort1)
DeletePort (for ReplyPort2)
Task3
CreatePort (for ReplyPort1)
CreatePort (for ReplyPort2)
CreateStdIO (for first I/O request)
CreateStdIO (for last I/O request)
Task3 could create one or more tasks using CreateTask
DeleteStdIO (for first I/O request)
DeleteStdIO (for last I/O request)
DeletePort (for ReplyPort1)
DeletePort (for ReplyPort2)
ReplyPort 1
ReplyPort 2
Device1
(Unit0)
Device1 I/O Port
BeginIO/DoIO/SendIO
Device2
(Unit0)
Device2 I/O Port
BeginIO/DoIO/SendIO
Device3
(Unit0)
Device3 I/O Port
BeginIO/DoIO/SendIO

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

Task1 contains two CreatePort statements to create two task reply ports for I/O requests coming back from Device1. In addition, Task1 contains a number of CreateStdIO function calls to create IOStdReq structures that send commands to Device1, 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, Task1 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_STOP, 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. QuickIO will be successful if the current system conditions allow it. The device internal routines will 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 QuickIO 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 QuickIO 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_ReplyPort 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 QuickIO, 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

All 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 SendIO. 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 SendIO 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.

## 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 these I/O request structures could be created by the Exec-support library functions CreateStdIO and CreateExtIO. Then only the I/O request structure Message substructure parameters 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 using 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 allocation process. (The figure is not intended to portray the RAM location of these structures.)

The initialization of IORequest0 is completed with a call to the OpenDevice function, which initializes its io_Device and io_Unit parameters. Other required IORequest0 parameters

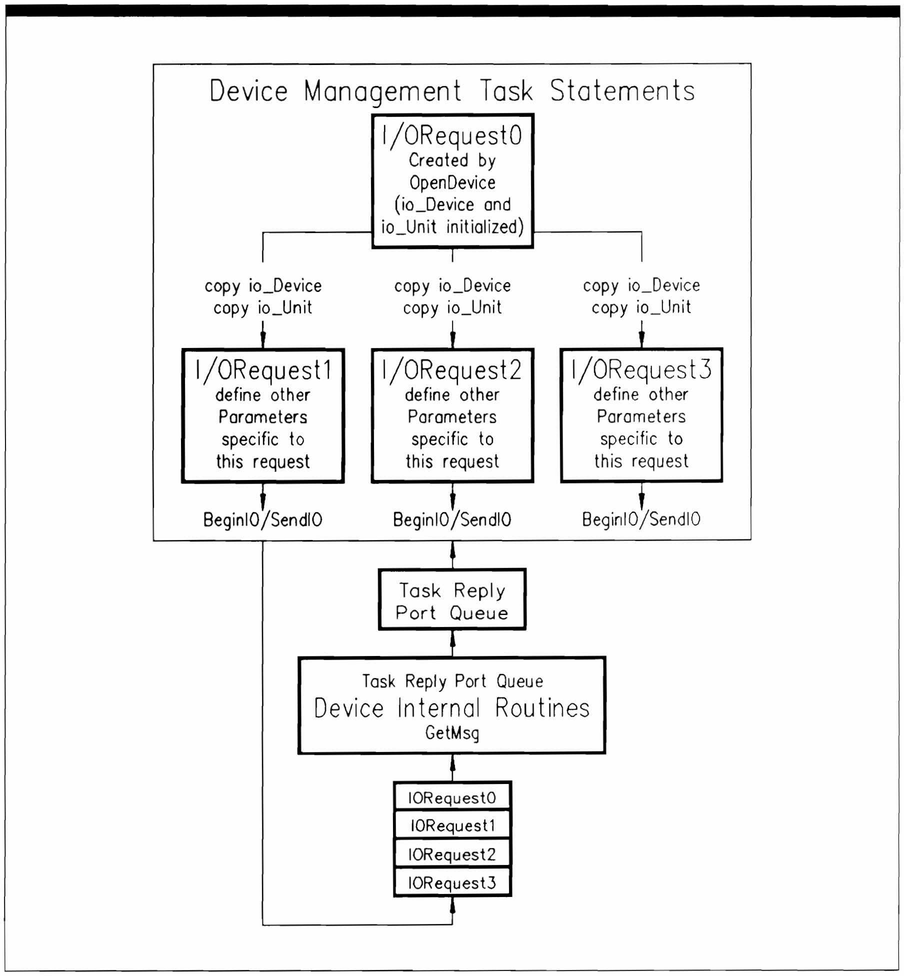


**Figure 2.5:** *Sending Multiple I/O Requests to a Device*

can be initialized using simple structure-parameter assignment statements. When all of these parameters are initialized, IORequest0 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, IORequest0 will be placed below them. The device internal routines will process IORequest0 when it gets to the top of the queue. However, if the system is busy with other tasks, devices, or interrupts, IORequest0 may have to wait in the queue until the system passes control to the device internal routines.

Once the task has dispatched IORequest0 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 IORequest0 to do so, because the device itself now owns IORequest0 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, IORequest1, 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 IORequest0, 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 IORequest0.

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 CheckIO 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 CheckIO 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, CheckIO, DoIO, and SendIO. Although these functions all end with "IO," 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 list 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 successful 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_DELEXP, 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, 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

| | |
|---|---|
| **iOReplyPort** | 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 DeleteExtIO. 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 ln_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 ioReplyPort 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 DeleteStdIO. 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 call 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 DeleteTask 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 CreateExtIO. 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 CreateExtIO function |
| **size** | The size of the extended I/O request structure as defined in CreateExtIO |

### 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 function 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 ln_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.

# 3 The Audio Device

## Introduction

This chapter discusses the Audio device, the most complicated device in the Amiga system. It is complex because multiple audio channels must be shared among multiple tasks, requiring an arbitration mechanism both to allocate channels and to allow channels to be stolen, while at the same time notifying other tasks of the current state of all four of the Audio device channels. Due to the Amiga's multitasking capabilities, the Audio device also allows for a double-buffered mode of operation, which further adds to its programming complexity.

Six functions can be used in programming the Audio device: AbortIO, BeginIO, OpenDevice, CloseDevice, AddDevice, and RemDevice. The DoIO and SendIO functions are not usually used in an Audio device management program. Instead, the BeginIO command is usually used to dispatch all Audio device commands, both asynchronous and synchronous I/O requests. DoIO and SendIO usually clear bits 4–7 of the IOAudio structure io_Flags parameter; BeginIO does not alter these bit values.

## Audio System Hardware

The Audio device hardware configuration is illustrated in Figure 3.1. Each audio channel has an 8-bit digital-to-analog converter driven by a DMA (direct memory access) channel. Each DMA channel can retrieve two data samples during each horizontal video-scanning line. The DMA can automatically play simple, steady tones, or a task can define and perform complex sound effects, including stereo effects and amplitude and frequency modulation effects produced by attaching audio channels to one another.

Sound data is always organized as 8-bit data items. Each item is a sample from a waveform stored in RAM. To conserve MEMF_CHIP RAM, a task normally defines only one cycle of the waveform in RAM. For an unchanging sound, the values at the waveform's beginning and end should be closely related to provide a smooth transition. This arrangement ensures that the repetition of the waveform sounds like a continuous stream of sound without pops and other audible discontinuities. The first byte of the audio channel data must always be on a word boundary in RAM, because the Audio device always retrieves one word (two bytes) at a time when playing the data.

The volume of sound produced by an audio channel ranges from 0 to 64, an arbitrary volume scale. These volume values correspond to decibel levels. For a typical volume output at level 64, with maximum data values ranging from $-128$ to 127, the maximum voltage output range available to an external audio amplifier lies between $+0.4$ and $-0.4$ volts.

The pitch of the sound produced by a waveform depends on its frequency, specified by the sampling period. The sampling period is the number of system clock ticks (timing intervals) that should elapse between each byte of audio data (each sample) fed to the converter for a specified audio channel. It is defined by the IOAudio structure ioa_Period parameter in an Audio device I/O request.

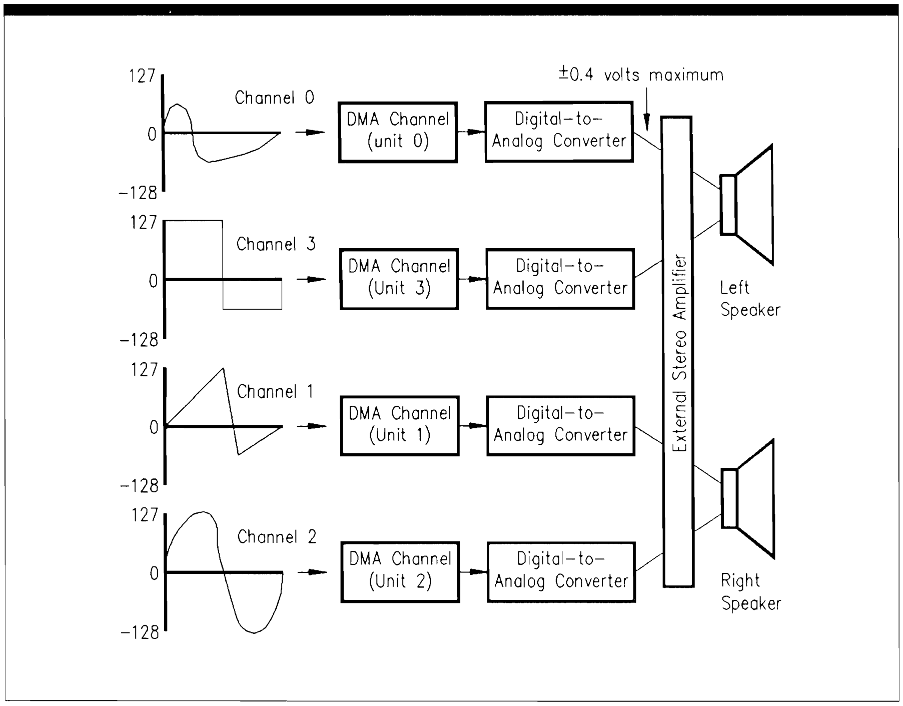


**Figure 3.1:** *Hardware Configuration Used by the Audio Device*

Each audio channel also has a period register, the current value of which is used as a countdown value; each time the period register counts down to 0, another sample is retrieved from the waveform data set. The value in the period register represents clock ticks per sample. The maximum period value a task can use is 65,535 ticks per sample, and the minimum is 124. A low value corresponds to high-frequency sound; a high value corresponds to low-frequency sound. If the period value is below 124, by the time the cycle count has reached 0, the audio DMA will not have had enough time to retrieve the next data sample and the previous sample will automatically be reused.

Because the Amiga system is a multitasking system, many tasks can be competing for the four available channels. Therefore, at any given time, a task may ask for one or more audio channels currently in use by another task. The Audio device system must decide which task gets each of the four audio channels at any particular time.

The Audio device channels are either allocated or free at a given point in time. Therefore, if a task wants one or more channels, the system can decide which channels should be allocated to the requesting task based on the current system state. For this reason, the actual assignment of channels to tasks can be left to the system; it is not the direct responsibility of the task. However, it is up to the task to characterize the relative importance of its channel needs with respect to other tasks. A task does this using two mechanisms: the channel combination array and channel allocation precedence.

## The Channel Combination Array

The channel combination array is the task's statement to the system about what audio channels it wants. The numbers in this array are used with the information provided in channel allocation precedence; together, these parameters allow the system to decide how to satisfy the task's audio channel request.

Table 3.1 illustrates the audio channel capabilities. Because there are four channels, there are 16 possible channel combinations, including all single- and multiple-channel allocations. These combinations are represented as 4-digit numbers, with the numeral 1 in any position where a channel is allocated and 0 in any position where a channel is not allocated. The number 0000 indicates that no channel is allocated, whereas 1111 indicates that all channels are allocated. Unit 0 is represented by the rightmost position in this number, and unit 3 is represented by the leftmost position. The system assigns unit 0 and unit 3 to the left speaker and unit 1 and unit 2 to the right speaker, as you saw in Figure 3.1.

The channel combination array's input to the ADCMD_ALLOCATE command or the OpenDevice function is determined by these associations. For example, if you want to play sound from the left speaker, you pass to the ADCMD_ALLOCATE command or OpenDevice function a channel combination array consisting of at least one of the 12 numbers that are valid for left-speaker assignments. The more decimal values a task specifies in the channel combination array, the greater are the chances that the audio channel (in this case, the left speaker) will be allocated to that task.

If your task needed to restrict its allocation choices further, it would specify fewer values in the channel combination array. For example, if you needed to play sounds from the left speaker only, using either channel 0 or channel 3 but not both, you would specify a channel combination array consisting of two entries: 1 and 8. Neither of these two combinations would produce any sound in the right speaker. You would specify 2 for the IOAudio structure ioa_Length parameter and then point ioa_Data to the array of four numbers representing the desired channel allocation sequence. This would be passed to the ADCMD_ALLOCATE command or OpenDevice function. The Audio device internal routines would work with this request, automatically comparing it to the current state of channel allocations.

On the other hand, if your task needed to play sounds from the left and right speakers at the same time to create a stereo effect and you did not care how it was accomplished by the system, you would set the channel combination array to nine ioa_Length values, as follows: 3, 5, 7, 10, 11, 12, 13, 14, and 15. The system would automatically compare this statement to the current state of all channels in the system, look at the requested allocation priority (ln_Pri parameter), and then allocate channels accordingly.

Your task could also attach channels in order to accomplish frequency or amplitude modulation. To do so, you would specify a channel combination array having three entries at most: 3, 6, or 12.

Keep in mind that the channel combination array is used in the following fashion to satisfy the task's requested allocation:

- The first entry in the channel combination array is used to see if the channels it specifies can be allocated. If they can, the system allocates them to the task and

**Table 3.1:** *Amiga Audio Channel Capabilities*

| Channel 3 | Channel 2 | Channel 1 | Channel 0 | Channel Combination Number | Left Speaker | Right Speaker | Stereo Sound | Amplitude or Frequency Modulation |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | — | — | — | — |
| 0 | 0 | 0 | 1 | 1 | ✓ | — | — | — |
| 0 | 0 | 1 | 0 | 2 | — | ✓ | — | — |
| 0 | 0 | 1 | 1 | 3 | ✓ | ✓ | ✓ | 0,1 attached |
| 0 | 1 | 0 | 0 | 4 | — | ✓ | — | — |
| 0 | 1 | 0 | 1 | 5 | ✓ | ✓ | ✓ | — |
| 0 | 1 | 1 | 0 | 6 | — | ✓ | — | 1,2 attached |
| 0 | 1 | 1 | 1 | 7 | ✓ | ✓ | ✓ | — |
| 1 | 0 | 0 | 0 | 8 | ✓ | — | — | — |
| 1 | 0 | 0 | 1 | 9 | ✓ | — | — | — |
| 1 | 0 | 1 | 0 | 10 | ✓ | ✓ | ✓ | — |
| 1 | 0 | 1 | 1 | 11 | ✓ | ✓ | ✓ | — |
| 1 | 1 | 0 | 0 | 12 | ✓ | ✓ | ✓ | 2,3 attached |
| 1 | 1 | 0 | 1 | 13 | ✓ | ✓ | ✓ | — |
| 1 | 1 | 1 | 0 | 14 | ✓ | ✓ | ✓ | — |
| 1 | 1 | 0 | 1 | 15 | ✓ | ✓ | ✓ | — |

copies the first entry of that array into the IOAudio structure io_Unit parameter; io_Unit then represents the Audio device units currently assigned to the task.

- If the first value for the channel combination array is not successful at allocating channels, the system looks at each of the other values in turn until it is either successful or unsuccessful. If it finds a value that produces a successful allocation, it sets the io_Unit parameter.
- If the system exhausts all values for the channel combination array and cannot allocate the requested channels, it tries to steal channels. It takes the ln_Pri parameter specified in the IOAudio structure and compares it to the known channel precedences of channels already allocated to other tasks. It then finds the channel combination array that requires it to steal the lowest-precedence channels from other tasks in the system.

## Channel Allocation Precedence

Channel allocation precedence is the priority you assign to an audio channel. It is always specified as the IOAudio structure ln_Pri parameter (a number between −128 and 127) you initialize before you call the ADCMD_ALLOCATE command or the OpenDevice function to allocate channels.

An ln_Pri value of −128 means that the task's request for a channel or channels is low priority; a value of 127 means that the task wants the channels allocated immediately. For example, if you wanted to play an urgent sound (an alert to the user, for example) from the left speaker, you would set ln_Pri at 127 and set the channel combination array for the left speaker. If you wanted to play a sound that did not need to be heard immediately from the right speaker, you would set ln_Pri at a much lower value and set the channel combination array for the right speaker.

If the allocation fails after all channel combination array values are tried with multiple channel requests, you could try to split the request into several single-channel requests, each with its own ln_Pri channel precedence parameter. When the task resubmits the request as several requests instead of one, the allocation may succeed.

In addition to using the ln_Pri parameter, you can tell the system that you want a channel or channels immediately by using the ADIOF_NOWAIT flag parameter, one of several possible values for the IOAudio structure io_Flags parameter that you initialize before dispatching an Audio device command. It tells the system to give the task the channel or channels it requests immediately—even if it has to steal them from another task. ADIOF_NOWAIT is described in later sections of this chapter.

### Channel Locking and Stealing

The ADCMD_LOCK command allows a task to lock an audio channel, which is necessary only if the task needs to work directly with the audio hardware registers. ADCMD_LOCK has two purposes: to notify a task that one or more of its channels has been stolen by a higher-precedence allocation in another task; and to prevent another task from stealing channels if the channel precedence assigned by another task is lower than that of the present task.

A task needs to know that its channels are about to be stolen so that it can clean up before letting the system give those channels to another task, and so that it does not try to use that channel until it is again available. Therefore, if you want to work directly with the audio channel registers and to prevent tasks in the system from experiencing allocation conflicts, always execute ADCMD_LOCK immediately after you allocate one or more channels; copy ioa_AllocKey into a new IOAudio structure that defines the upcoming ADCMD_LOCK command, and then dispatch the ADCMD_LOCK I/O request.

When a task tries to allocate channels that are also allocated by the present task, the second task's ADCMD_ALLOCATE command will be temporarily suspended until the ADCMD_LOCK command is replied (with io_Error equal to ADIOERR_CHANNELSTOLEN) to the first task. The first task will then know that one or more of its allocated channels is about to be stolen; it can clean up before those channels are actually stolen. Once the cleanup is complete, the system executes an ADCMD_FREE command automatically in the first task. The second task can then proceed with its allocation of the channel it just stole.

Never make the freeing of a stolen channel dependent on the allocation of another channel—that sequence may cause a system deadlock. To keep a channel from being stolen, set the channel precedence ln_Pri parameter to its maximum value (127). Never use an ADCMD_LOCK command to effectively impose the highest channel precedence; use the ADCMD_SETPREC command for that purpose.

### Informing a Task of a Sound

You may want the system to inform a task when an audio channel begins to play a sound. Once notified, the task can branch to other activities (for example, graphics) that should occur simultaneously with sound production. The Audio device system provides the IOAudio structure ioa_WriteMsg Message substructure for this purpose. Set the IOAudio structure io_Flags parameter to ADIOF_WRITEMESSAGE if you want the task to know when the Audio device internal routines start playing the waveform defined by a CMD_WRITE command. The system signals the task when this message arrives in the task reply-port queue.

Note that this Message substructure is distinct from the other IOAudio structure Message substructure. One Message structure is part of the IORequest substructure and is used to send the I/O request to the task reply-port queue when the CMD_WRITE command is completed. The second one, ioa_WriteMsg, is at the end of the IOAudio structure. It is used to signal the task when the Audio device internal routines start to execute a CMD_WRITE command. The task can take appropriate additional action consistent with that knowledge. Always remember that the io_Flags ADIOF_WRITEMESSAGE bit is used only for the CMD_WRITE command.

## The Allocation Key

Recall that for some Amiga devices, device-unit sharing among tasks is requested by setting the I/O request structure io_Flags parameter SHARED bit when the device is opened. In particular, the Serial and Parallel devices work this way. If this flag parameter bit is not specified, the device will usually be opened in exclusive access mode, the default for opening most Amiga devices with OpenDevice.

The Audio device does not use this arrangement. Instead, it uses a number called the allocation key, a two-byte parameter in the IOAudio structure. The specific value of the allocation key (ioa_AllocKey) is determined by a direct call to the ADCMD_ALLOCATE command or by an OpenDevice function call if the ioa_Length parameter specified in that call is not 0.

It is important to remember that the ioa_AllocKey parameter is not the channel combination array and it is not a 4-digit binary number representing allocated channels. Instead, it is an internally defined value that the Audio device software system defines and recognizes in various contexts. It summarizes, for system use only, the known state of a task's allocated channels for one task in the system.

Each task that deals with the Audio device has its own ioa_AllocKey parameter, which can change each time that task executes an ADCMD_ALLOCATE command or an OpenDevice function call. Each additional command that a task sends to the Audio device routines must specify the current ioa_AllocKey parameter.

When working with the Audio device, you must copy the ioa_AllocKey parameter from one IOAudio structure into another IOAudio structure. The first IOAudio structure represents the request replied by an ADCMD_ALLOCATE command or OpenDevice function call; the second represents another command about to be dispatched. This is similar to copying io_Device and io_Unit parameters between I/O request structures for other Amiga devices (see Chapter 2).

The ioa_AllocKey parameter represents the channels that were actually allocated to satisfy a task's allocation request. The system has done its best to allocate channels without stealing (if possible) or by stealing only the lowest-priority channels currently owned by other tasks. Once the system provides the ioa_AllocKey parameter, the task knows which channels the system decided to allocate and can work with those channels to send other Audio device commands.

For example, to play some sounds out of a speaker, you initialize an IOAudio structure to represent a CMD_WRITE command request. As part of the structure initialization, you initialize the ioa_AllocKey parameter. To get the correct value for ioa_AllocKey, you copy the ioa_AllocKey value from the IOAudio structure replied by the original ADCMD_ALLOCATE or OpenDevice function call, using simple C language structure-parameter assignment statements to do so.

## Audio Device Commands

To program the Audio device, you can use any of eight standard commands and seven device-specific commands. Here are some important points about the commands:

- All Audio device commands use the IOAudio structure to send a request to the Audio device internal routines. The IOAudio structure is an IORequest structure with an appended set of device-specific parameters.

- Most of the Audio device commands are either synchronous commands or asynchronous commands; however, there are four Audio device commands that can operate in both modes: CMD_WRITE, ADCMD_ALLOCATE, ADCMD_LOCK, and ADCMD_WAITCYCLE.
- QuickIO is possible for all of the Audio device commands, regardless of whether the command is synchronous or asynchronous. You should generally dispatch both asynchronous and synchronous I/O requests with the BeginIO function. For asynchronous requests, you can use the CheckIO, WaitIO, GetMsg, and Remove functions in the usual manner.
- I/O requests may be queued, processed as QuickIO, or processed automatically in immediate mode. All of the Audio device commands (except for CMD_WRITE) occur immediately when dispatched by a task. Obvious exceptions to this are ADCMD_FINISH and ADCMD_PERVOL when the SYNCCYCLE flag is set, and ADCMD_WAITCYCLE when there is a CMD_WRITE in progress.
- Some Audio device commands can be used in interrupt code. However, they can only be used below interrupt level 5.
- Most Audio device commands and functions return error values in the IOAudio structure io_Error parameter.
- Some Audio device commands and functions change specific IOAudio structure parameters when they return.

## Sending Commands to the Audio Device

Figure 3.2 shows how commands are sent to Audio device internal routines. The lines with arrows represent the parameters you should initialize and those returned by the Audio device internal routines.

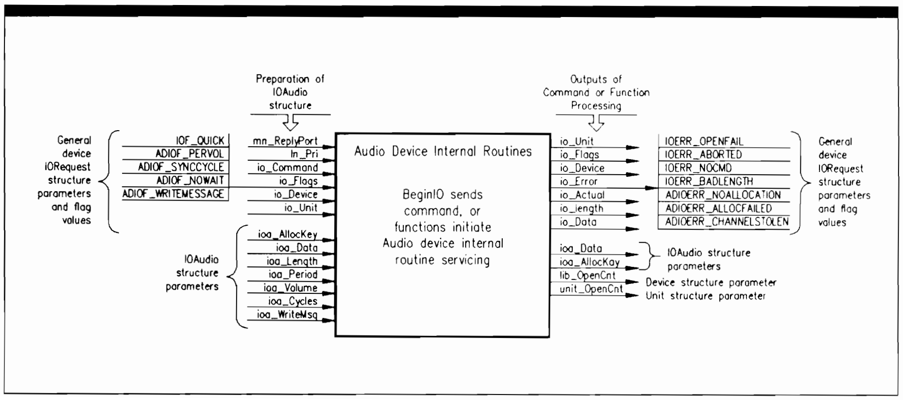


**Figure 3.2:** *Audio Device Command and Function Processing*

As Figure 3.2 shows, the Audio device programming process consists of three phases:

1. IOAudio structure preparation. Here, you initialize parameters in the IOAudio structure in preparation for sending a command to the Audio device internal routines. These parameters include the general device-request parameters and all device-specific IOAudio structure parameters. These parameters provide an information path to the data needed by the Audio device internal routines to process the command.
2. Device processing. The only part you play in this phase is to send the command to the Audio device using the BeginIO function; control passes to the device and system internal routines.
3. Command output processing. In this phase, the system and device internal routines have complete control over the values found in the parameters. The results of command processing have been returned to the task that originally issued the command. Note that the Audio device does not return any values for the io_Actual, io_Length, or io_Device parameter. The parameters returned provide an information path to the data needed by the requesting task.

Figure 3.2 also depicts the parameters that play a part in function setup and processing for the Audio device. The OpenDevice and CloseDevice functions affect the Device structure lib_OpenCnt parameter and Unit structure Unit_OpenCnt parameter; OpenDevice also affects the IOAudio structure ioa_AllocKey parameter.

## Structures for the Audio Device

The Audio device works with two structures: IOAudio and AudChannel, as shown in Figure 3.3. You do not usually deal with the AudChannel structure directly; instead, you let the system define it for you when you specify parameters in the IOAudio structure. Therefore, most function and command operations deal with the IOAudio structure and its IORequest substructure parameters. For more detailed control of the Audio device routines, you can also initialize AudChannel structure parameters directly. Note that the AudChannel structure is a substructure inside the Custom structure (see the Hardware/Custom.h INCLUDE file).

### The IOAudio Structure

The IOAudio structure looks like this:

```
struct IOAudio {
   struct IORequest ioa_Request;
   WORD ioa_AllocKey;
   UBYTE *ioa_Data;
   ULONG ioa_Length;
   UWORD ioa_Period;
```

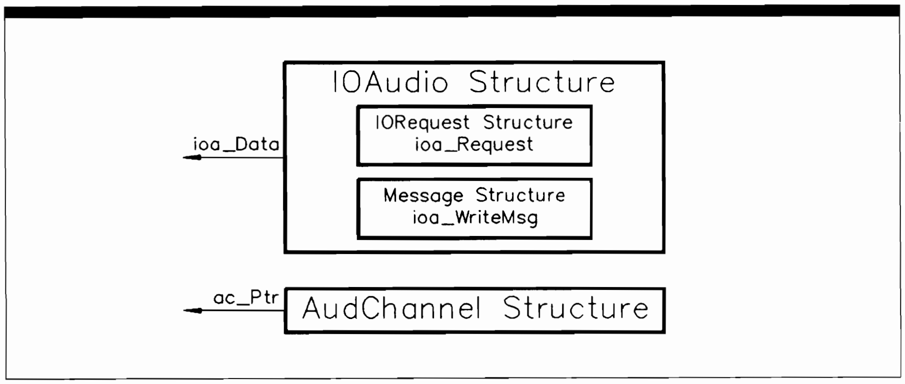


**Figure 3.3:** *Audio Device Structures*

```
    UWORD ioa_Volume;
    UWORD ioa_Cycles;
    struct Message ioa_WriteMsg;
};
```

The parameters in the IOAudio structure have the following meanings:

- ioa_Request. This parameter is the name of an IORequest substructure whose six parameters help represent a command sent to the Audio device. The IORequest structure contains a Message substructure, which contains the mn_ReplyPort parameter, which points to the task reply-port queue MsgPort structure.
- ioa_AllocKey. This parameter is the allocation key. It is generated by the ADCMD_ALLOCATE command and the OpenDevice function call. It is then used by most of the other Audio device commands and functions. To perform multiple-channel commands, all channels must have the same allocation key, even when they are not allocated simultaneously. To use a key you already have, copy its value into this parameter for the current IOAudio structure. ADCMD_ALLOCATE only returns a new and unique allocation-key value if you initialize this parameter to 0 before the ADCMD_ALLOCATE command is dispatched.
- ioa_Data. This parameter is a pointer to the data for the Audio device request. It takes on different interpretations depending on the specific function or command. For example, for the CMD_WRITE command, io_Data is a pointer to a waveform array consisting of signed bytes in chip-addressable memory.
- ioa_Length. This parameter represents the data length of some item related to a particular Audio device I/O request. It takes on different interpretations depending on the specific function or command. For example, for the CMD_WRITE command, ioa_Length is the number of bytes in the waveform array (an even number from 2 to 131,072).

- ioa_Period. This parameter is always the period of the Audio device I/O request, measured in system clock ticks; it ranges from 127 to 6,553. The anti-aliasing filter (which eliminates unwanted harmonics) works below 300 or 500, depending on the waveform. This parameter is used as input to the ADCMD_PERVOL and CMD_WRITE commands only.
- ioa_Volume. This parameter is always the volume of sound you want to produce with the Audio device I/O request; it ranges from 0 to 64 and is used as input to the ADCMD_PERVOL and CMD_WRITE commands only. The system will use the default value (64) if you do not specify otherwise.
- ioa_Cycles. This parameter is the number of cycles in the Audio device request. It tells the Audio device routines the number of times to play the audio waveform array data. This parameter is used as input to the CMD_WRITE command only. If ioa_Cycles is set to 0, the system default value, the waveform will repeat indefinitely.
- ioa_WriteMsg. This parameter is the name of a Message substructure, the last structure entry in the IOAudio structure, which represents an extra message-passing capability used by the CMD_WRITE command only. This parameter is used only when the IOAudio structure io_Flags ADIOF_WRITEMESSAGE parameter is set; if so set, the Audio device internal routines will notify the dispatching task that CMD_WRITE execution has begun. Do not confuse this Message substructure with the other Message substructure in IORequest, which serves as the reply mechanism for all Audio device commands.

The IOAudio structure IORequest substructure io_Flags parameter bit values are as follows:

- ADIOF_PERVOL. Set this flag if you want to set the period and volume at the start of CMD_WRITE execution. If it is not set, the previous (or default) period and volume will be used.
- ADIOF_SYNCCYCLE. Set this flag if you want the current waveform to finish playing the current cycle before the task takes additional action (for example, before aborting that waveform I/O request with the AbortIO function). The change can take place immediately or at the end of the current cycle; it can be used to produce vibratos, glissandos, tremolos, and volume envelopes in music, as well as to change the volume of the sound.
- ADIOF_NOWAIT. Set this flag if you want to produce a sound immediately and you cannot wait to free an Audio device channel for allocation. It will cause the ADCMD_ALLOCATE command to return an ADIOERR_ALLOCFAILED error if the system cannot allocate any of the channels without waiting for one to become free. The Audio device will continue to try the allocation request whenever one of the four channels is freed until it is successful. To cancel the request, use AbortIO.
- ADIOF_WRITEMESSAGE. Set this flag if you want the system to tell your task when a sound starts playing.

### The AudChannel Structure

The AudChannel structure works together with the CMD_WRITE command. It contains a set of parameters to represent the waveform data necessary to play sound through an audio channel. Under most circumstances, the system internally defines the AudChannel parameters. However, for more direct control of the audio channels, a task can work with the AudChannel directly.

The AudChannel structure looks like this:

```
struct AudChannel {
   UWORD ac_Ptr;
   UWORD ac_Len;
   UWORD ac_Per;
   UWORD ac_Vol;
   UWORD ac_Dat;
   UWORD ac_Pad[2];
};
```

The parameters in the AudChannel structure are as follows:

- ac_Ptr. This is a pointer to the audio channel waveform data.
- ac_Len. This is the length of the audio channel waveform data in words (two bytes per word).
- ac_Per. This is the sampling period to be assigned to the waveform data.
- ac_Vol. This is the volume to be assigned to the waveform data.
- ac_Dat. This is a pair of waveform data samples. The resolution of each sample is 1 part in 256; each sample ranges from −128 to 127.
- ac_Pad[2]. This is two bytes of data to word-align the AudChannel structure.

## Audio Device Error Codes

The error messages returned by the Audio device are as follows:

- ADIOERR_NOALLOCATION. An audio channel could not be allocated to satisfy an allocation request. If a task attempts to perform an allocation on an already stolen channel, this bit is set and the bit in the IORequest structure io_Unit parameter corresponding to the stolen channel is cleared so that the task knows which channel has been stolen.
- ADIOERR_ALLOCFAILED. The attempted allocation of the channel failed. If the ADIOF_NOWAIT flag bit is set, the system will return this error if none of the audio channels could be allocated.
- ADIOERR_CHANNELSTOLEN. The audio channel was stolen by another task.

The Amiga device system provides four errors that are common to all devices:

- IOERROR_OPENFAIL. An OpenDevice function call failed because the system could not open the device for some reason. A common reason for both disk- and ROM-resident devices is that there is not enough memory available to open the device library; the solution is usually to free some memory and call OpenDevice again. In addition, for disk-resident devices the cause may be that the appropriate device library file was not on the disk in the DEVICES: directory.
- IOERR_ABORTED. The task intentionally aborted the request using either the AbortIO function or the CMD_FLUSH command. Once task conditions are properly set up, the task can reset the io_Error parameter to 0 and dispatch the I/O request again.
- IOERR_NOCMD. The dispatching task specified an I/O request io_Command parameter value that the device internal routines did not understand. The task should redefine that parameter and dispatch that command again.
- IOERR_BADLENGTH. The dispatching task specified an I/O request io_Length parameter value that the device internal routines did not understand. The task should redefine that parameter and dispatch that command again.

The precise definition of these errors appears in the Exec/Errors.h INCLUDE file. The error names shown there represent individual values ($-1$ to $-4$) that the system might assign to a replied IORequest structure io_Error parameter. Most individual devices have a set of io_Error values that start at numerical value 0. See the individual device INCLUDE files (Audio.h, etc.) for the names and numerical definitions of these device-specific errors.

## USE OF FUNCTIONS

## *CloseDevice*

### Syntax of Function Call

```
CloseDevice (iOAudio)
             A1
```

### Purpose of Function

This function closes access to the Audio device internal routines for one or more Audio device units. When it returns, the IOAudio structure io_Device pointer will be set to $-1$.

In addition, if there are channels allocated with the same allocation key, CloseDevice will free them. CloseDevice sets the io_Unit parameter unit bit of any free channels to 0; if all channels are closed, all four low-order bits of the io_Unit parameter will be 0 when CloseDevice returns.

CloseDevice decrements the Unit structure unit_OpenCnt parameter for that unit. If unit_OpenCnt is reduced to 0 for all open units and a deferred expunge sent by this or another task is pending, the Audio device structures are expunged from RAM as soon as all open units are closed.

### Inputs to Function

| | |
|---|---|
| **iOAudio** | A pointer to an IOAudio structure; also a pointer to an IORequest structure |

### Preparation of the IOAudio Structure

The IOAudio structure is defined by a previous OpenDevice function call.

### Discussion

CloseDevice terminates access to a set of device routines for a particular Audio device unit. The io_Unit parameter of the IORequest substructure specifies the units affected.

If your task has different allocation keys for the channels, you cannot use Close-Device to close all of them at once. Instead, you must issue one ADCMD_FREE command for each unique allocation key your task is using. Then your task can call CloseDevice to close all units.

If a number of tasks have opened an Audio device unit in shared access mode, the Unit structure unit_OpenCnt parameter will reflect the number of tasks that have opened but not closed that unit. Once all units are closed, the Audio device routines can be expunged from the system.

## *OpenDevice*

### Syntax of Function Call

```
error = OpenDevice ("audio.device" unitNumber, iOAudio, 0)
D0                  A0             D0          A1       D1
```

## Purpose of Function

This function opens access to the Audio device routines for one or more units. If successful, it sets the IORequest structure io_Device pointer to point to a Device structure that the system uses to manage the Audio device units. In addition, if the IOAudio structure ioa_Length parameter is other than 0, OpenDevice will execute the ADCMD_ALLOCATE command indirectly to allocate audio channels, thereby setting io_Unit to point to the set of Unit structures that manages the device request message ports for each allocated unit. OpenDevice also increments the Device (Library) structure lib_OpenCnt parameter, thereby preventing a deferred expunge.

OpenDevice requires a properly initialized task reply port with a task signal bit number allocated. If unsuccessful, OpenDevice returns an error value in the IOAudio (IORequest) structure io_Error parameter. It does not wait for the allocation to succeed, and it closes the Audio device.

Here are the values returned by OpenDevice:

- io_Device. This is a pointer to a Device structure, which manages the successfully opened units of the Audio device. The Device structure contains the information necessary to reach all the data and routines in the Audio device library.
- io_Unit. This is a bitmap of the successfully allocated channels; bits 0–3 correspond to channels 0–3. This value allows a task to determine which units were actually allocated by the OpenDevice call. It will always be 0 if the ioa_Length parameter was 0.
- io_Error. This value indicates the state of attempted opens and allocations. A 0 here indicates that the requested channel allocations succeeded.

## Inputs to Function

| | |
|---|---|
| **"audio.device"** | A pointer to a null-terminated string representing the name of the Audio device |
| **unitNumber** | The Audio device unit number (0–3) for the unit or units to open; the IOAudio structure io_Unit parameter specifies the units affected. Specify this argument only if you want to both open the Audio device and allocate units at the same time; specify 0 to open the Audio device without allocating any units. |
| **iOAudio** | A pointer to an IOAudio structure |
| **0** | Indicates that the flags argument is not used for the Audio device |

### Preparation of the IOAudio Structure

Initialize the following parameters:

- ln_Pri. Set this only if you want to both open the Audio device and allocate a unit (channel), and only if the ioa_Length parameter is not 0.
- mn_ReplyPort. Set this to point to a MsgPort structure for the task reply port that will receive the reply from the Audio device internal routines when it has finished processing OpenDevice. This parameter is only necessary if you want to allocate as well as open an Audio device unit. The requesting task will ascertain from the I/O request reply which units have or have not been allocated. You will set this parameter only if the ioa_Length parameter is not 0.
- ioa_Data. Set this to point to a channel combination array (only if the ioa_Length parameter is not 0).
- ioa_Length. Set this to the length of the channel combination array, a value from 0 to 15. Use a 0 here if you do not want to allocate any units.

### Discussion

OpenDevice is used to open the Audio device routines for access by a task. When OpenDevice returns, the IOAudio structure io_Device parameter will point to a Device structure used to manage one or more Audio device units; each open unit will have a unique Unit structure.

## STANDARD DEVICE COMMANDS

## *CMD_CLEAR*

### Purpose of Command

CMD_CLEAR is a multiple-channel command. It clears all Audio device internal buffers for each channel specified by a set bit in io_Unit, if the ioa_AllocKey parameter is correct. If the allocation key (ioa_AllocKey) is not correct, CMD_CLEAR returns an error value (ADIOERR_NOALLOCATION).

The CMD_CLEAR command is always treated as a synchronous I/O request and only replies to the task reply-port queue if the io_Flags IOF_QUICK bit is cleared. The results of command execution are as follows:

- io_Unit. This is a 4-bit bitmap of the channels successfully cleared; bits 0–3 correspond to channels 0–3.

- io_Error. A 0 here indicates that the command was successful. ADIOERR_NO-ALLOCATION indicates that the IOAudio structure ioa_AllocKey parameter does not match the current allocation key for the channel.

### Preparation of the IOAudio 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 Unit structures that manage each addressed Audio device unit. Set io_Command to CMD_CLEAR. Set io_Flags to 0 if not used. Otherwise, initialize it to IOF_QUICK for QuickIO, which always succeeds for this command.

### Discussion

Two standard device commands directly affect the internal device buffers of the Audio device: CMD_UPDATE and CMD_CLEAR. CMD_CLEAR restores the device internal routines to a known state without resetting the entire Audio device system with CMD_RESET.

## *CMD_FLUSH*

### Purpose of Command

CMD_FLUSH is a multiple-channel command. It flushes all pending I/O requests from a device request queue for a specified set of device units. It aborts any executing CMD_WRITE commands and any pending CMD_WRITE commands that are queued in any of the specified device units. For each channel specified by a set bit in io_Unit, if the ioa_AllocKey parameter is correct, CMD_FLUSH aborts all in-progress or queued writes and any ADCMD_WAITCYCLE I/O requests waiting to synchronize with the end of the cycle.

CMD_FLUSH is always treated as a synchronous I/O request and only replies to the task reply-port queue if the IOF_QUICK flag parameter is cleared. Do not use CMD_FLUSH in interrupt code at interrupt level 5 or higher.

The results of command execution are as follows:

- io_Unit. This is a 4-bit bitmap of the channels successfully flushed. Bits 0–3 correspond to channels 0–3.

- io_Error. 0 indicates that the command was successful. ADIOERR_NOALLOCATION indicates that the IOAudio structure ioa_AllocKey parameter does not match the current allocation key for the channel.

### Preparation of the IOAudio 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 Unit structures that manage each addressed Audio device unit. Set io_Command to CMD_FLUSH. Set io_Flags to 0 if not used. Otherwise, initialize it to IOF_QUICK for QuickIO, which always succeeds for this command.

### Discussion

Two Audio device commands directly affect the device request queues of a set of audio channels: CMD_RESET and CMD_FLUSH. CMD_RESET calls CMD_FLUSH indirectly. The AbortIO function can also remove a specified I/O request structure from an Audio device-unit request queue.

Because CMD_FLUSH is a very destructive command, you normally use it only if you want to restore the system to some known state—for example, to remove all pending and active I/O requests from a set of one or more device-unit I/O request queues.

## *CMD_READ*

### Purpose of Command

CMD_READ is a single-channel command. For each channel specified by a bit set in io_Unit, if the ioa_AllocKey parameter is correct, CMD_READ returns a pointer to the IOAudio structure for the CMD_WRITE command currently writing on the selected channel. If there is no CMD_WRITE command in progress on the specified unit, CMD_READ returns a 0 value.

CMD_READ is always treated as a synchronous I/O request and only replies if the io_Flags IOF_QUICK bit is cleared. The results of command execution are as follows:

- io_Unit. This is a 4-bit bitmap of the channels successfully read; bits 0–3 correspond to channels 0–3.

- io_Error. 0 indicates that the command was successful. ADIOERR_NOALLOCATION indicates that the IOAudio structure ioa_AllocKey parameter does not match the current allocation key for the channel.

- ioa_Data. This points to an IOAudio structure representing the currently executing CMD_WRITE command on the specified unit. It is 0 if no write is currently in progress on the specified unit.

### Preparation of the IOAudio 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 Unit structures that manage each addressed Audio device unit. Set io_Command to CMD_READ. Set io_Flags to 0 if not used. Otherwise, initialize it to IOF_QUICK for QuickIO, which always succeeds for this command.

### Discussion

Five commands directly affect or relate to a currently executing Audio device CMD_WRITE command: ADCMD_PERVOL, ADCMD_WAITCYCLE, CMD_STOP, ADCMD_FINISH, and CMD_READ. Most of the Audio device commands have either a constructive or a destructive effect on the state of the Audio device system. CMD_READ, however, does nothing more than return a pointer to an IOAudio structure representing the currently active CMD_WRITE command.

If a task needs to know whether an audio channel is being used before it attempts to write to it, that task can dispatch CMD_READ for the channel. Then, if any task in the system is currently using the channel to play a sound, the CMD_READ request structure replied to the calling task will contain a pointer to the IOAudio structure representing the CMD_WRITE command. The calling task can then look at the ioa_Data parameter in the replied CMD_READ command IOAudio structure to obtain a pointer to the CMD_WRITE IOAudio structure.

## *CMD_RESET*

### Purpose of Command

CMD_RESET is a multiple-channel command. For each channel specified by a bit set in io_Unit, if the ioa_AllocKey parameter is correct, CMD_RESET clears the audio hardware registers and channel-to-channel attach bits set for frequency or amplitude modulation. It also sets the audio interrupt vector, cancels all pending I/O requests for all of the specified audio channels, and restarts the channels if they are currently stopped by the CMD_STOP command.

CMD_RESET is always treated as a synchronous I/O request and only replies if the io_Flags IOF_QUICK bit is cleared. Do not use CMD_RESET in interrupt code at interrupt level 5 or higher.

The results of command execution are as follows:

- io_Unit. This is a 4-bit bitmap of the channels successfully reset; bits 0–3 correspond to channels 0–3.
- io_Error. 0 indicates that the command was successful. ADIOERR_NOALLOCATION indicates that the IOAudio structure ioa_AllocKey parameter does not match the current allocation key for the channel.

### Preparation of the IOAudio 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 Unit structures that manage each addressed Audio device unit. Set io_Command to CMD_RESET. Set io_Flags to 0 if not used. Otherwise, initialize it to IOF_QUICK for QuickIO, which always succeeds for this command.

### Discussion

Two Audio device commands directly affect the device request queue for a set of audio channels: CMD_RESET and CMD_FLUSH. CMD_FLUSH flushes all pending I/O requests from a set of one or more device-unit I/O request queues.

The CMD_RESET command is very destructive—among other things, it calls CMD_FLUSH indirectly to flush the queued I/O requests. CMD_RESET also calls CMD_START to start any channels stopped previously with CMD_STOP; when a task once again starts to send requests to the specified Audio device units, there is no need to restart the channels with an explicit CMD_START command.

## *CMD_START*

### Purpose of Command

CMD_START is a multiple-channel command. For each channel specified by a bit set in io_Unit, if the ioa_AllocKey parameter is correct and the channel was previously stopped by CMD_STOP, CMD_START starts all pending CMD_WRITE commands to the channel.

CMD_START starts multiple channels simultaneously to minimize distortion if the channels are playing the same waveform and their outputs are mixed. It is always treated as a synchronous I/O request and only replies if the io_Flags IOF_QUICK bit is cleared. Do not use CMD_START in interrupt code at interrupt level 5 or higher.

The results of command execution are as follows:

- io_Unit. This is a 4-bit bitmap of the channels successfully started; bits 0–3 correspond to channels 0–3.
- io_Error. A 0 here indicates that the command was successful. ADIOERR_NOALLOCATION indicates that the IOAudio structure ioa_AllocKey parameter does not match the current allocation key for the channel.

## Preparation of the IOAudio 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 Unit structures that manage each addressed Audio device unit. Set io_Command to CMD_START. Set io_Flags to 0 if not used. Otherwise, initialize it to IOF_QUICK for QuickIO, which always succeeds for this command.

## Discussion

Two commands start and stop the production of sound in Audio device units: CMD_START and CMD_STOP. CMD_STOP stops the sound produced by executing a CMD_WRITE command for any of a set of specified audio channels.

CMD_START is similar to the Ctrl-Q command used to restart screen output on most computers—it restarts execution of a CMD_WRITE command stopped previously by CMD_STOP, just as Ctrl-Q restarts screen output stopped previously with Ctrl-S. If a set of channels was previously stopped by a single CMD_STOP command, CMD_START will restart all of them simultaneously, provided the CMD_START IOAudio structure io_Unit parameter is the same as the CMD_STOP command IOAudio structure io_Unit parameter.

## *CMD_STOP*

## Purpose of Command

CMD_STOP is a multiple-channel command. For each channel specified by a bit set in io_Unit, if the ioa_AllocKey parameter is correct, CMD_STOP immediately stops any executing CMD_WRITE commands in progress on the channel. Once a channel is stopped, the system automatically queues writes to that channel until the CMD_START command restarts that channel or CMD_RESET resets it.

CMD_STOP is always treated as a synchronous I/O request and only replies if the io_Flags IOF_QUICK bit is cleared. Do not use CMD_STOP in interrupt code at level 5 or higher.

The results of command execution are as follows:

- io_Unit. This is a 4-bit bitmap of the channels successfully stopped; bits 0–3 correspond to channels 0–3.
- io_Error. 0 indicates that the command was successful. ADIOERR_NOALLOCATION indicates that the IOAudio structure ioa_AllocKey parameter does not match the current allocation key for the channel.

### Preparation of the IOAudio 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 Unit structures that manage each addressed Audio device unit. Set io_Command to CMD_STOP. Set io_Flags to 0 if not used. Otherwise, initialize it to IOF_QUICK for QuickIO, which always succeeds for this command.

### Discussion

Two commands start and stop the production of sound in Audio device units: CMD_START and CMD_STOP. CMD_START restarts the sound produced by an executing CMD_WRITE in any of a set of specified audio channels previously stopped with CMD_STOP. The CMD_STOP command is similar to the Ctrl-S command used for screen output on most computers—it stops specified currently executing CMD_WRITE commands at the earliest possible opportunity.

Note that CMD_STOP does not provide for an IOAudio structure io_Flags ADIOF_SYNCCYCLE flag parameter bit. Therefore, if a task needs to stop playing a sound at the end of the current cycle for a set of Audio device channels, it should dispatch the ADCMD_WAITCYCLE command just before it dispatches CMD_STOP for those channels.

## CMD_UPDATE

### Purpose of Command

CMD_UPDATE is a multiple-channel command. It forces all Audio device internal device buffers out to the Audio device hardware. For each channel specified by a bit set in io_Unit, if the ioa_AllocKey parameter is incorrect, CMD_UPDATE returns an error value (ADIOERR_NOALLOCATION).

CMD_UPDATE is always treated as a synchronous I/O request and only replies to the task reply-port queue if the io_Flags IOF_QUICK bit is cleared.

The results of command execution are as follows:

- io_Unit. This is a 4-bit bitmap of the channels successfully updated; bits 0–3 correspond to channels 0–3.
- io_Error. 0 indicates that the command was successful. ADIOERR_NOALLOCATION indicates that the IOAudio structure ioa_AllocKey parameter does not match the current allocation key for the channel.

### Preparation of the IOAudio 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 Unit structures that manage each addressed Audio device unit. Set io_Command to CMD_UPDATE. Set io_Flags to 0 if not used. Otherwise, initialize it to IOF_QUICK for QuickIO, which always succeeds for this command.

### Discussion

There are two standard device commands that directly affect the internal device buffers of the Audio device: CMD_UPDATE and CMD_CLEAR. CMD_CLEAR clears the internal device buffers for a set of one or more Audio device channels. CMD_UPDATE writes the current contents out from all Audio device internal device buffers, thereby writing (playing) the information bytes currently present in those buffers. This is similar to the operation of the TrackDisk device, which forces the floppy- or hard-disk track buffers out to the physical disk unit during a power failure.

## *CMD_WRITE*

### Purpose of Command

CMD_WRITE is a single-channel command. For the channel specified by io_Unit, if the ioa_AllocKey parameter is correct, CMD_WRITE plays a sound for the channel and queues I/O requests if there is another write in progress or if the channel has been stopped by CMD_STOP.

CMD_WRITE is always treated as an asynchronous I/O request if there is no error; it clears IOF_QUICK and replies the I/O request after it has finished writing. If there is an error, CMD_WRITE is treated as synchronous and only replies if the io_Flags IOF-_QUICK bit is cleared. CMD_WRITE replies after it completes execution. Do not use CMD_WRITE in interrupt code at interrupt level 5 or higher.

The results of command execution are as follows:

- io_Unit. This is a 4-bit bitmap of the channels successfully written; bits 0–3 correspond to channels 0–3.
- io_Error. 0 indicates that the command was successful. ADIOERR_NOALLOCATION indicates that the IOAudio structure ioa_AllocKey parameter does not match the current allocation key for the channel. ADIOERR_ABORTED indicates that the Audio device I/O request has been aborted. ADIOERR_CHANNELSTOLEN indicates that the channel has been stolen.

## Preparation of the IOAudio 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 Unit structures that manage each addressed unit of the Audio device. Set io_Command to CMD_WRITE.

Also initialize the following command-specific parameters:

- io_Flags. Set this to 0 if not used. Otherwise, initialize it to ADIOF_PERVOL to load a new period and volume value for the channel. Initialize io_Flags to ADIOF_WRITEMESSAGE to reply the CMD_WRITE request at the start of the write operation using the ioa_WriteMsg Message structure to pass the message to the sending task reply port.
- ioa_Data. Set this to a point to a waveform data array, which consists of signed integer bytes that range from −128 to 127. Each waveform data point must be in chip-addressable RAM and word-aligned.
- ioa_Length. Set this to the length (in bytes) of the waveform data array. This must be an even number between 2 and 131,072.
- ioa_Period. Set this to a new value for the sampling period. The sampling period is measured in 279.365 nanoseconds intervals; the sampling rate can be from 127 to 65,536. The anti-aliasing (harmonic elimination) filter works below 300 or 500 cycles per second depending on the waveform used. Specify this parameter if ADIOF_PERVOL is set.
- ioa_Volume. Set this to a new value for the volume of the sound. Volume ranges from 0 to 64 in a linear fashion. Specify this parameter only if ADIOF_PERVOL is set.
- ioa_Cycles. Set this to the number of times to repeat the waveform array; this can be from 0 to 65,535. Use 0 if you want to repeat the waveform an infinite number of times.

- ioa_WriteMsg. Initialize this to the name of the Message structure containing the message to send to the task reply port at the start of CMD_WRITE execution. This message is only replied if the io_Flags ADIOF_WRITEMESSAGE bit is set.

### Discussion

CMD_WRITE is the only Audio device command that directly plays information out to the Amiga external hardware. Any task can dispatch a continuous stream of CMD_WRITE commands to each Audio device unit allocated to it. Each of these CMD_WRITE commands will be queued in the device-unit request queue for those units; there is no QuickIO mechanism for the CMD_WRITE command. The information in the task-defined buffers specified by CMD_WRITE commands will always be played in the order in which the CMD_WRITE command I/O requests were queued.

However, this does not mean that once the CMD_WRITE requests are queued, the operation of the system cannot be altered—CMD_FLUSH, AbortIO, and CMD_RESET alter the specific I/O requests queued to each unit. Also, the CMD_START and CMD-_STOP commands can start and stop the playing of buffer information through each device unit. Finally, the ADCMD_PERVOL command can always change the period and volume of the queued request when CMD_WRITE actually starts playing information through each of the specified channels.

## DEVICE-SPECIFIC COMMANDS

## *ADCMD_ALLOCATE*

### Purpose of Command

ADCMD_ALLOCATE tries to allocate a group of audio channels. If the length of the channel combination array is specified as 0 in the ioa_Length parameter, the allocation will succeed. Otherwise, the ADCMD_ALLOCATE command checks each combination specified by the channel combination array, one at a time and in the specified order, and tries to allocate one of the channel combinations.

If ADCMD_ALLOCATE must steal channels, it uses the channel combination that steals the lowest-precedence channel; it cannot steal a channel of equal or greater precedence.

The ADCMD_ALLOCATE command replies only if the io_Flags IOF_QUICK bit is cleared; otherwise, the allocation is treated as an asynchronous I/O request, thereby clearing IOF_QUICK and replying the request after the allocation is completed. A task should not use the ADCMD_ALLOCATE command in interrupt code at any interrupt level.

The results of command execution are as follows:

- io_Unit. This is a 4-bit bitmap of the successfully allocated channels; bits 0–3 correspond to channels 0–3.
- io_Flags. The IOF_QUICK flag parameter is cleared if the Audio device treats the ADCMD_ALLOCATE command as an asynchronous request.
- io_Error. 0 indicates that the command was successful; ADIOERR_ALLOCFAILED indicates that the requested allocation failed. ADIOERR_NOALLOCATION indicates that the IOAudio structure ioa_AllocKey parameter does not match the current allocation key for the channel.

## Preparation of the IOAudio Structure

Initialize ln_Pri, the required channel's allocation precedence, to a value from −128 to 127. 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 Unit structures that manage each addressed Audio device unit. Initialize io_Command to ADCMD_ALLOCATE.

Also initialize the following command-specific parameters:

- io_Flags. Set this to 0, or set it to IOF_QUICK for QuickIO. If QuickIO fails, the Audio device routines will treat ADCMD_ALLOCATE as an asynchronous I/O request; the system will reply the I/O request and clear the IOF_QUICK flag bit. Initialize io_Flags to ADIOF_NOWAIT if you want ADCMD_ALLOCATE to return ADIOERR_ALLOCFAILED if the allocation fails on the first try.
- ioa_AllocKey. Set this to 0 if you want processing of ADCMD_ALLOCATE to generate a new value for the allocation key. Otherwise, ioa_AllocKey must be copied from the IOAudio structure replied by OpenDevice or the most recent ADCMD_ALLOCATE command processed.
- ioa_Data. Set this to point to a channel combination array; bits 0–3 correspond to channels 0–3.
- ioa_Length. Set this to the length of the channel combination array; it can have a value from 0 to 15. A value of 0 always succeeds.

## Discussion

There are two ways to allocate audio channels. The first is to use the OpenDevice function, which calls the ADCMD_ALLOCATE command indirectly. The second way is to use the ADCMD_ALLOCATE command directly.

ADCMD_ALLOCATE can allocate one or more audio channels to the calling task. It will try not to steal audio channels; however, if it must, it will always steal the lowest-precedence channels. In addition, ADCMD_ALLOCATE cannot steal an audio channel whose current precedence is greater than or equal to the IOAudio structure ln_Pri parameter representing the ADCMD_ALLOCATE command.

If the allocation is successful, ADCMD_ALLOCATE checks to see if any of the channels have been locked by ADCMD_LOCK. If one or more have been locked, ADCMD-_ALLOCATE replies the original IOAudio structure representing ADCMD_LOCK to the task reply port with the io_Error parameter set to ADIOERR_CHANNELSTOLEN. It then places the I/O request in a list waiting for the locked channels.

When all channels required to satisfy it have been unlocked, ADCMD_ALLOCATE calls CMD_RESET. It then generates a new ioa_AllocKey value for the newly allocated channels if the previous ioa_AllocKey parameter was 0; otherwise, it allocates the same value. It goes on to copy ioa_AllocKey and ln_Pri into the IOAudio structure for each of the allocated channels and copies the 4-bit channel bitmap into the io_Unit parameter of the IOAudio structure for each of them.

ADCMD_ALLOCATE can operate either synchronously or asynchronously, depending on conditions in the system when it is dispatched. It operates synchronously if the allocation succeeds and did not require any locked channels to be stolen, or if the allocation fails and the ADIOF_NOWAIT flag parameter bit is set. ADCMD_ALLOCATE only replies the I/O request if the IOF_QUICK flag parameter bit is cleared. Otherwise, it operates asynchronously, clearing the IOF_QUICK flag parameter bit and replying the I/O request when the allocation is finished.

If ADCMD_ALLOCATE must steal audio channels to satisfy the allocation request, all device commands issued in other tasks from which these channels were stolen will return ADIOERR_CHANNELSTOLEN when those tasks try to dispatch their commands. Unless channels are stolen, a task must always free (ADCMD_FREE) all allocated channels when it is finished using them before another task can use them.

If you decide to work directly with the audio hardware registers in assembly language, you must either immediately lock (ADCMD_LOCK) the channels that ADCMD-_ALLOCATE allocates, or set the channel precedence to a maximum value (127) to prevent the channels from being stolen by another task.

## ADCMD_FINISH

### Purpose of Command

ADCMD_FINISH aborts the currently executing CMD_WRITE command for a set of specified channels; it is a multiple-channel command. For each selected channel as specified by io_Unit, if the allocation key is correct and there is a CMD_WRITE in progress, ADCMD_FINISH aborts the current write immediately or at the end of the current cycle if ADIOF_SYNCCYCLE is set.

The ADCMD_FINISH command is a synchronous I/O request and only replies if the IOF_QUICK flag parameter is cleared. Do not use ADCMD_FINISH in interrupt code at interrupt level 5 or higher.

The results of command execution are as follows:

- io_Unit. This is a 4-bit bitmap of the successfully finished channels; bits 0-3 correspond to channels 0-3.
- io_Error. 0 indicates that the command was successful. ADIOERR_NOALLOCATION indicates that IOAudio structure ioa_AllocKey parameter does not match the current allocation key for the channel.

### Preparation of the IOAudio 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 Unit structures that manage each addressed Audio device unit. Initialize io_Command to ADCMD_FINISH.

Set io_Flags to 0 if not used. Otherwise, initialize it to IOF_QUICK for QuickIO; if QuickIO fails, the Audio device internal routines will treat ADCMD_FINISH as an asynchronous I/O request. The system will reply the I/O request and the IOF_QUICK flag bit will be cleared. Initialize io_Flags to ADIOF_SYNCCYCLE if you want ADCMD_FINISH to finish at the end of the current cycle for all units specified.

### Discussion

Five commands directly affect a currently executing CMD_WRITE command: ADCMD_PERVOL, ADCMD_WAITCYCLE, CMD_READ, CMD_STOP, and ADCMD_FINISH. ADCMD_FINISH allows a task to finish writing data to (playing sound on) one or more audio channels. If the specified channels are not currently writing, ADCMD_FINISH has no effect on the system. The write can be aborted immediately or at the end of the current cycle of the waveform data; this choice is controlled by the ADIOF_SYNCCYCLE io_Flags bit. If ADIOF_SYNCCYCLE is set, the sound can finish smoothly, thus providing a clean transition for the human ear.

The AbortIO function can also abort an ongoing CMD_WRITE command. However, because it is designed to abort all types of requests, AbortIO is not tied specifically to aborting CMD_WRITE commands and cannot synchronize the abort operation with the end of a write cycle.

## ADCMD_FREE

### Purpose of Command

ADCMD_FREE is a multiple-channel command. For each channel specified by io_Unit, if the allocation key is correct, ADCMD_FREE restores the channel to a known state using CMD_RESET and changes the channel's allocation key parameter. It then makes the channel available for reallocation by an ADCMD_ALLOCATE command or Open-Device function call.

ADCMD_FREE unlocks a channel if it is locked by the ADCMD_LOCK command. It clears the channel bit for the channel designated in the IOAudio structure io_Unit parameter that represents the original ADCMD_LOCK request. If the IOAudio structure for the ADCMD_LOCK request has no channel bits set in its io_Unit parameter, ADCMD_FREE replies the original ADCMD_LOCK request. It then checks if there are any allocation requests waiting for the channel.

ADCMD_FREE is always a synchronous I/O request and only replies if the IOF-_QUICK is cleared. Do not use ADCMD_FREE in interrupt code at any interrupt level.

The results of command execution are as follows:

- io_Unit. This is a 4-bit bitmap of the successfully freed channels; bits 0–3 correspond to channels 0–3.
- io_Error. 0 indicates that the command was successful. ADIOERR_NOALLOCATION indicates that the IOAudio structure ioa_AllocKey parameter does not match the current allocation key for the channel.

### Preparation of the IOAudio 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 Unit structures that manage each addressed Audio device unit. Set io_Command to ADCMD_FREE. Set io_Flags to 0 if not used. Otherwise, initialize it to IOF_QUICK for QuickIO, which always succeeds for this command.

### Discussion

The ADCMD_FREE command allows a task to free audio channels it once allocated with the OpenDevice function or the ADCMD_ALLOCATE command. Once these channels are freed by the task, it or another task can allocate those channels with OpenDevice or ADCMD_ALLOCATE.

Note that ADCMD_FREE unlocks channels previously locked by the ADCMD_LOCK command. In fact, ADCMD_FREE provides the only direct way—short of the destructive CMD_RESET command—to unlock previously locked channels.

## *ADCMD_LOCK*

### Purpose of Command

ADCMD_LOCK is a multiple-channel command. It prevents audio channels from being stolen from the task that issues it. For each channel specified by io_Unit, if the ioa_AllocKey parameter is correct, the ADCMD_LOCK command locks the channel, thereby preventing subsequent allocations from stealing it.

The ADCMD_LOCK command is treated as an asynchronous I/O request if ioa_AllocKey is correct, in which case it clears the IOF_QUICK bit and eventually replies to the task reply-port queue. Otherwise, ADCMD_LOCK is treated as synchronous, and only replies if IOF_QUICK is cleared. Do not use ADCMD_LOCK in interrupt code at any interrupt level.

The results of command execution are as follows:

- io_Unit. This is a 4-bit bitmap of the successfully locked channels; bits 0–3 correspond to channels 0–3.
- io_Error. 0 indicates that the command was successful. ADIOERR_NOALLOCATION indicates that the IOAudio structure ioa_AllocKey parameter does not match the current allocation key for the channel. ADIOERR_CHANNELSTOLEN indicates that a locking operation is attempting to lock a stolen channel.
- io_Flags. The IOF_QUICK bit is cleared if ioa_AllocKey is correct.

## Preparation of the IOAudio 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 Unit structures that manage each addressed Audio device unit. Set io_Command to ADCMD_LOCK. Set io_Flags to 0.

## Discussion

ADCMD_LOCK allows a task to prevent other tasks from stealing the audio channels it allocates with OpenDevice or ADCMD_ALLOCATE. Unlike setting the channel precedence to its maximum value with ADCMD_ALLOCATE, OpenDevice, or ADCMD-_SETPREC, which then causes all subsequent allocations by other tasks to fail, using ADCMD_LOCK causes all higher-precedence allocations, even no-wait (ADIOF_NOWAIT) allocations, to wait until the requested channels are unlocked. It does not prevent other tasks from eventually getting those channels; it merely delays those allocations.

Locked channels can only be unlocked by executing ADCMD_FREE, which clears the io_Unit parameter channel bits for each channel freed. ADCMD_LOCK does not reply until all the channels it locks are freed, unless a higher precedence allocation attempts to steal one of the locked channels.

If a channel is successfully stolen, ADCMD_LOCK replies and sets the IOAudio structure io_Error parameter to ADIOERR_CHANNELSTOLEN. In this case, the channels should be freed as soon as possible. To avoid deadlock, a task should never make the freeing of channels dependent on the completion of another task's allocation.

## ADCMD_PERVOL

### Purpose of Command

ADCMD_PERVOL is a multiple-channel command. It changes the period and volume for writes currently in progress on selected audio channels. For each channel specified by io_Unit, if ioa_AllocKey is correct and there is CMD_WRITE in progress, ADCMD-_PERVOL loads a new volume and period either immediately or at the end of the current cycle, depending on ADIOF_SYNCCYCLE.

The ADCMD_PERVOL command is always treated as a synchronous I/O request and only replies to the task reply-port queue if IOF_QUICK is cleared. Do not use ADCMD_PERVOL in interrupt code at interrupt level 5 or higher.

The results of command execution are as follows:

- io_Unit. This is a 4-bit bitmap of the channels that were successfully loaded with new values for the period and volume; bits 0–3 correspond to channels 0–3.
- io_Error. 0 indicates that the command was successful. ADIOERR_NOALLOCATION indicates that the IOAudio structure ioa_AllocKey does not match the current allocation key for the channel.

### Preparation of the IOAudio 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 Unit structures that manage each addressed unit of the Audio device. Set io_Command to ADCMD-_PERVOL.

Also initialize the following command-specific parameters:

- io_Flags. Set this to 0 if not used. Otherwise, initialize it to IOF_QUICK for QuickIO, which always succeeds for ADCMD_PERVOL. Set io_Flags to ADIOF-_SYNCCYCLE if you want ADCMD_PERVOL to change the period and volume at the end of the current write cycle; otherwise, it will change the period and volume immediately.
- ioa_Period. This is a new value for the sampling period. The sampling period is measured in 279.365-nanosecond intervals; the sampling rate can be from 127 to 65,536. The anti-aliasing (harmonic elimination) filter works below 300 or 500 Hz, depending on the waveform used.
- ioa_Volume. This is a new value for the volume of the sound. Volume ranges from 0 to 64.

## Discussion

Five commands directly affect a currently executing CMD_WRITE command: ADCMD-_FINISH, ADCMD_WAITCYCLE, CMD_READ, CMD_STOP, and ADCMD_PERVOL. ADCMD_PERVOL changes the period and volume for a currently executing CMD_WRITE. It allows a task to finish writing data to (playing sound on) one or more audio channels before the period and volume are changed. If the specified channels are not currently writing, ADCMD_PERVOL has no effect on the system.

The period and volume can be changed immediately or at the end of the current CMD_WRITE. The choice is controlled by the io_Flags ADIOF_SYNCCYCLE bit in the ADCMD_PERVOL command IOAudio structure; if this bit is set, the period and volume will not be changed until the end of the current cycle, thus providing a clear transition for the sound.

## *ADCMD_SETPREC*

### Purpose of Command

ADCMD_SETPREC changes the channel precedence for a set of Audio device channels. It is a multiple-channel command. For each channel specified by io_Unit, if the allocation key is correct, ADCMD_SETPREC changes the allocation precedence to a new value. It also checks if there are any allocation requests waiting for the channel, which now has a higher precedence.

The ADCMD_SETPREC command is always treated as a synchronous I/O request and only replies if the IOF_QUICK bit is cleared. Do not use ADCMD_SETPREC in interrupt code at any interrupt level.

The results of command execution are as follows:

- io_Unit. This is a 4-bit bitmap of the channels successfully set or changed; bits 0–3 correspond to channels 0–3.
- io_Error. 0 indicates that the command was successful. ADIOERR_NOALLOCATION indicates that the IOAudio structure ioa_AllocKey parameter does not match the current allocation key for the channel.

### Preparation of the IOAudio 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 Unit structures that manage each addressed Audio device unit. Set io_Command to ADCMD_SET-PREC. Set io_Flags to 0 if not used. Otherwise, initialize it to IOF_QUICK for QuickIO, which always succeeds for this command.

### Discussion

At any given time, all allocated channels can have distinct channel allocation precedences established by previous executions of OpenDevice, ADCMD_ALLOCATE, or ADCMD-_SETPREC. You can use the ADCMD_SETPREC command to fine-tune the current channel allocation precedences in a task. By doing so, you can control the channel arbitration process among the various tasks in the system as precisely as you wish.

## *ADCMD_WAITCYCLE*

### Purpose of Command

The ADCMD_WAITCYCLE command causes the dispatching task to wait for the Audio device hardware to complete the current waveform cycle of a CMD_WRITE command, which allows the current note to finish playing. It is a single-channel command. For the channel specified by io_Unit, if the ioa_AllocKey parameter is correct and there is a CMD_WRITE command in progress for the channel, ADCMD_WAITCYCLE does not reply until the end of the current cycle. If there is no CMD_WRITE currently in progress, CMD_WRITE replies immediately.

ADCMD_WAITCYCLE is treated as an asynchronous request only if it is waiting for a cycle to complete; otherwise, it is treated as a synchronous request and replies only if IOF_QUICK is cleared. Do not use ADCMD_WAITCYCLE in interrupt code at interrupt level 5 or higher.

The results of command execution are as follows:

- io_Unit. This is a 4-bit bitmap of the channels that successfully waited for a cycle to complete; bits 0–3 correspond to channels 0–3.
- io_Flags. The IOF_QUICK bit is cleared if a write is in progress on the selected channel.
- io_Error. 0 indicates that the command was successful. ADIOERR_NOALLOCATION indicates that the IOAudio structure ioa_AllocKey parameter does not match the current allocation key for the channel. ADIOERR_ABORTED is returned if the I/O request has been aborted or ADIOERR_CHANNELSTOLEN if the channel has been stolen by another task.

## Preparation of the IOAudio 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 Unit structures that manage each addressed Audio device unit. Set io_Command to ADCMD_WAITCYCLE.

Set io_Flags to 0 if not used. Otherwise, initialize it to IOF_QUICK for QuickIO. If QuickIO fails, ADCMD_ALLOCATE will be treated as an asynchronous I/O request; the system will reply the I/O request and clear the IOF_QUICK bit.

## Discussion

Five commands directly affect a currently executing CMD_WRITE command: ADCMD-_PERVOL, CMD_READ, CMD_STOP, ADCMD_FINISH, and ADCMD_WAIT-CYCLE. ADCMD_WAITCYCLE allows a task to cooperate with other tasks and to synchronize its activities with other events in the Audio device system. Because ADCMD_WAITCYCLE is a single-channel command, it is dispatched for one audio channel at a time, allowing any next step in a task sequence to be coordinated with the end of a write cycle for a specific channel. This allows you to fine-tune the timing and sequence of Audio device tasks and programs.

Note that the ADCMD_WAITCYCLE request is only replied if there is a write in progress for the specified channel and if the cycle of that write has completed.

# 4

# The Narrator Device

## Introduction

This chapter discusses the Narrator device's functions and commands and the Translator library's Translate function. Here you learn what is required to make the Amiga work with text files composed of English text strings and how to convert them to phoneme strings, producing spoken words that can be heard out of any speakers connected to the Amiga. The Narrator device is programmed by a few Exec functions and six commands, all of which are standard device commands designed specifically for the Narrator device. Also of importance here is the relationship between the Narrator device and the Audio device.

## English-String Processing

Figure 4.1 shows an English string being processed by the Translator library, Narrator device internal routines, and Audio device internal routines. This diagram is a guideline for such tasks as designing a talking word-processing program.

A task places the English string into a task-defined text buffer. The Translator library translates the English string into a phoneme string equivalent on a word-for-word basis using the Translator library built-in Voice Synthesis library. To do so, the Translator library must consult a word-exception table built into its routines. Words not in that table are translated literally. You could also design your program to handle the exceptions by specifying your own word-exception table and producing corresponding phoneme strings directly. The result is a series of phonemes in a phoneme-string buffer.

The Narrator device can then do two things. It can instruct the Audio device internal routines to speak the phoneme string aloud using data in the Narrator_rb structure. This is defined as a CMD_WRITE operation. Additionally, it can instruct the Graphics library routines to display changing mouth shapes using data in the Mouth_rb structure. This is defined as a CMD_READ operation.

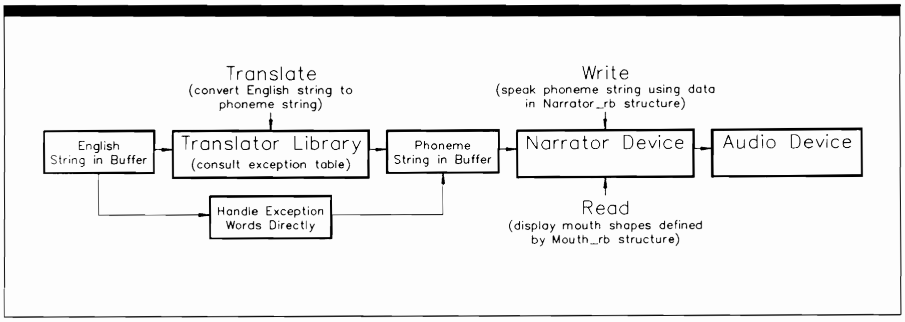

Translate
(convert English string to phoneme string)
Write
(speak phoneme string using data in Narrator_rb structure)
English String in Buffer
Translator Library
(consult exception table)
Phoneme String in Buffer
Narrator Device
Audio Device
Handle Exception Words Directly
Read
(display mouth shapes defined by Mouth_rb structure)

**Figure 4.1:** *Processing an English String*

## Narrator Device Commands

Table 4.1 presents a summary of the Narrator device commands. The Narrator device uses no device-specific commands, only standard device commands. Two of the Narrator device commands change parameters in the Narrator device-related structures, as shown in the table. These effects are detailed under the individual command and function discussions later in this chapter. Five of the Narrator device commands affect the io_Error parameter of the Narrator_rb structure IOStdReq substructure. None of the Narrator device commands support QuickIO.

### Sending Commands to the Narrator Device

Figure 4.2(a) depicts the general scheme used to send commands to the Narrator device internal routines. The lines with arrows do not represent pointer parameters; they represent the parameters you should initialize. In Figure 4.2(b), the arrows represent the parameters returned by the device internal routines.

The first phase is the structure preparation phase over which the programmer has complete control. The choice of parameters to initialize depends on the specific command you plan to send to the Narrator device. Always remember that if you do not explicitly initialize the Narrator_rb or Mouth_rb structure parameters before you dispatch a CMD_WRITE or CMD_READ command, their values will either be fixed by the most

**Table 4.1:** *Narrator Device Commands*

| Command | Quick I/O Possible? | Queued I/O Possible? | Values Affected | io_Error |
|---|---|---|---|---|
| CMD_FLUSH | No | No | — | Set to 0 |
| CMD_READ | No | Yes | width<br>height<br>shape | Set to error value |
| CMD_RESET | No | No | — | Set to 0 |
| CMD_START | No | No | — | Set to 0 |
| CMD_STOP | No | No | — | Set to 0 |
| CMD_WRITE | No | Yes | io_Actual | Set to error value |

previous value assigned or by the default value, depending on the specific programming sequence.

The second phase is the Narrator device internal routine processing phase. Here you send the command to the internal device routines using a BeginIO, DoIO, or SendIO device command-dispatching function. Control then passes to a mixture of the Narrator device, Audio device, and Exec system internal routines. The command I/O request will always get queued in the Narrator device request queue; the Narrator device internal routines do not support QuickIO.

The third phase is command or function output parameter processing. The system, Narrator, and Audio device internal routines have complete control over the values found in these structure parameters. The results of Narrator device command processing have been returned to the task that originally issued the command.

The outputs from device-command processing can be found in the parameters as shown in the figure. Notice that the io_Error value can take on one or more of the set of 19 values. Figures 4.2(a) and (b) also depict the parameters that play a part in Narrator device function setup and processing. Note that the Narrator_rb and Mouth_rb structure parameters shown in Figure 4.2(b) have been initialized to their default values by OpenDevice.

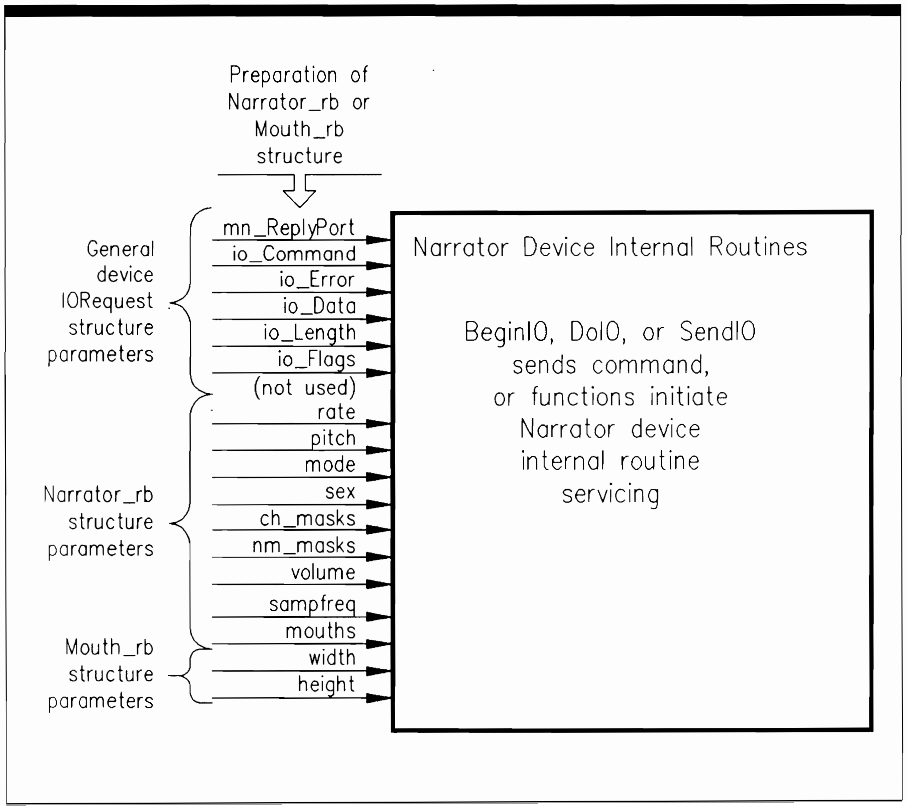

Preparation of Narrator_rb or Mouth_rb structure
General device IORequest structure parameters
Narrator_rb structure parameters
Mouth_rb structure parameters
mn_ReplyPort
io_Command
io_Error
io_Data
io_Length
io_Flags
(not used)
rate
pitch
mode
sex
ch_masks
nm_masks
volume
sampfreq
mouths
width
height
Narrator Device Internal Routines
BeginIO, DoIO, or SendIO sends command, or functions initiate Narrator device internal routine servicing

**Figure 4.2(a):** *Narrator Device Command and Function Processing (Specifications)*

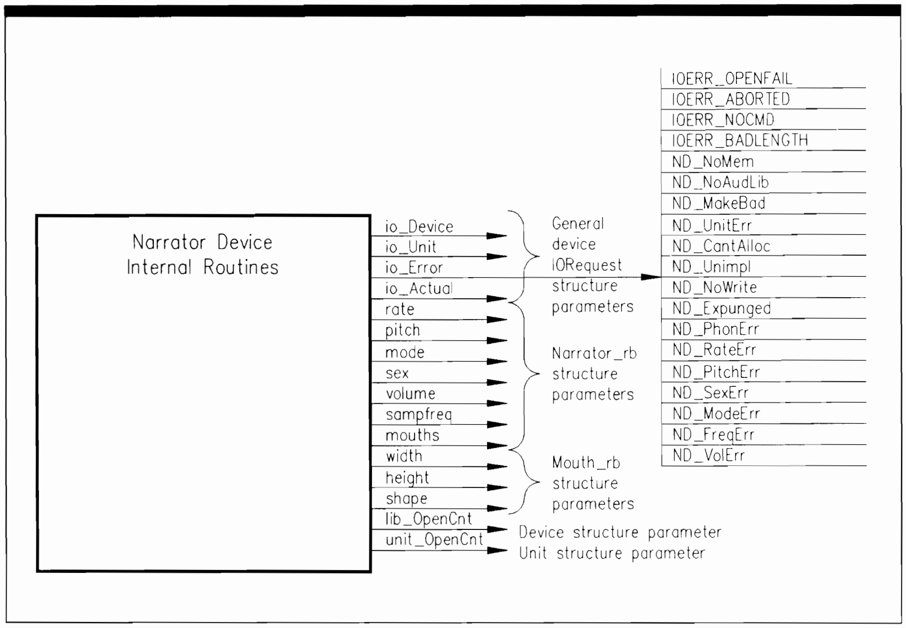


**Figure 4.2(b):** *Narrator Device Command and Function Processing (Outputs)*

## Structures for the Narrator Device

Figure 4.3 shows that the Narrator device deals with two structures: Narrator_rb and Mouth_rb. The Narrator_rb structure defines the audio output component of the narration either indirectly from the English string through the Translator library or directly from phoneme strings defined by a task. The Mouth_rb structure communicates display information to the Amiga display screen; it does not communicate audio data to the Audio device internal routines. The Narrator_rb and Mouth_rb structures are not directly linked. The Translator library has no programmer-accessible structures.

### The Narrator_rb Structure

The Narrator_rb structure contains an IOStdReq substructure named Message. It is used to define I/O requests passed back and forth between a task and the Narrator device internal routines. In particular, this substructure contains the mn_ReplyPort pointer parameter that points to a MsgPort structure representing the task reply-port queue of a task that dispatches a Narrator device command. The Narrator_rb structure also contains a pointer to a channel-combination array (ch_masks) that is used to define the desired set of audio channels.

The Narrator_rb structure is defined as follows:

```
struct narrator_rb {
   struct IOStdReq message;
```

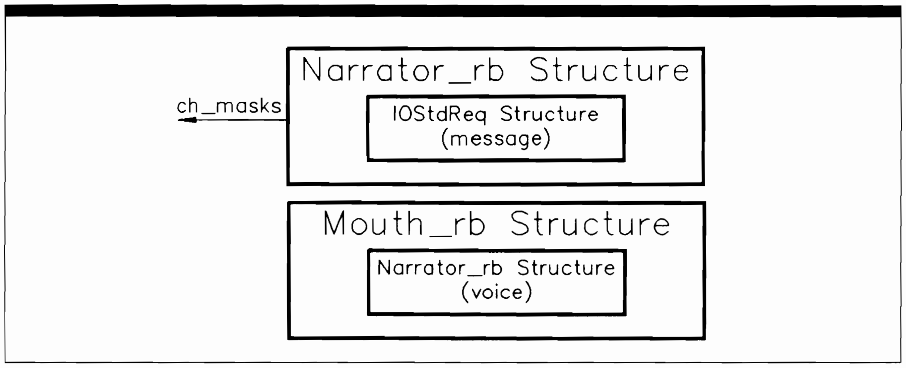


**Figure 4.3:** *Narrator Device Structures*

```
        UWORD rate;
        UWORD pitch;
        UWORD mode;
        UWORD sex;
        UBYTE *ch_masks;
        UWORD nm_masks;
        UWORD volume;
        UWORD sampfreq;
        UBYTE mouths;
        UBYTE chanmask;
        UBYTE numchan;
        UBYTE pad;
};
```

The parameters in the Narrator_rb structure have the following meanings:

- Message is the name of an IOStdReq substructure inside the Narrator_rb structure. The mn_ReplyPort parameter should be initialized to point to the MsgPort structure that represents the task reply-port queue for I/O requests.
- The rate parameter represents the speaking rate of the voice. It varies between 40 and 400 words per minute. The default is 150 words per minute.
- The pitch parameter represents the pitch of the voice. It is a baseline pitch, above and below which the actual speaking occurs. The baseline pitch can vary between 65 and 320; the default is 110.
- The mode parameter determines monotone (robotic) or natural expressive voice. The default mode is a natural expressive voice.
- The sex parameter determines the sex of the voice. The default is MALE.
- The ch_masks parameter is a pointer to a channel-combination array to be used in selecting Audio device channels for the voice.

- The nm_masks parameter contains the number of entries in the channel-combination array used by the Audio device internal routines to satisfy the I/O request.
- The volume parameter controls the volume of the voice. The volume can vary between 0 and 64. The default is 64, maximum volume.
- The sampfreq parameter is the sampling frequency used to sample the audio data associated with the I/O request. The sampling frequency can vary between 5,000 and 28,000; the default is 22,200 cycles per second.
- The mouths parameter should be set to 1 if the task needs to read and display mouth shapes using the Narrator device CMD_READ command while the system is executing a CMD_WRITE command. Otherwise, this parameter should be set to 0.
- The chanmask parameter is the Audio device's channel mask for the I/O request. It is for system internal use only.
- The numchan parameter represents the number of Audio device channels used. It is for system internal use only.
- The pad parameter is a one-byte padding used to word-align the structure data.

## The Mouth_rb Structure

The Mouth_rb structure contains a Narrator_rb substructure named Voice, which provides the detailed information required by Mouth_rb to define the shape of the mouth for each syllable of each word. It is usually a direct copy of the Narrator_rb structure for a previous CMD_WRITE command. In this way, the Mouth_rb structure is used for a series of CMD_READ commands corresponding to changing mouth shapes for a single CMD_WRITE command.

The Mouth_rb structure is defined as follows:

```
struct mouth_rb {
   struct narrator_rb voice;
   UBYTE width;
   UBYTE height;
   UBYTE shape;
   UBYTE pad;
};
```

The parameters in the Mouth_rb structure have the following meanings:

- The voice parameter is the name of a Narrator_rb substructure associated with the Mouth_rb structure. The Narrator_rb structure for each CMD_WRITE command should be copied into this substructure position.
- The width parameter represents the width of the mouth (in pixels) as returned by the Narrator device internal routines. It changes as CMD_READ commands return different values for the changing mouth shape.

- The height parameter represents the height of the mouth (in pixels) as returned by the Narrator device internal routines. It also changes as CMD_READ commands return different values.
- The shape parameter is the internal definition of the shape of the mouth as returned by the Narrator device internal routines. It also changes as CMD_READ commands return different values.
- The pad parameter is a one-byte padding to word-align the structure data.

## Narrator Device Error Codes

The error codes returned by the Narrator device have the following meanings:

- ND_NoMem. There is not enough memory to allocate the data and structures for the Narrator device internal routines and supporting structures.
- ND_NoAudLib. The system cannot find the Audio device library. The Audio device should be in WCS (Write Control Store) ROM after the KickStart disk is loaded.
- ND_MakeBad. There was an error in a MakeLibrary call resulting in a bad Narrator or Audio device library.
- ND_UnitErr. The task tried to allocate a Narrator device unit other than unit 0.
- ND_CantAlloc. The Audio device internal routines cannot allocate the number of audio channels specified in the Narrator_rb structure.
- ND_Unimpl. The task requested a currently unimplemented command.
- ND_NoWrite. The task tried to dispatch a CMD_READ without first dispatching a CMD_WRITE; or a CMD_WRITE finished execution, so there are no more mouth shapes for CMD_READ to read.
- ND_Expunged. The system cannot open the Narrator or Audio device libraries because one or both have their Library structure lib_Flags parameters currently set to LIBF-_DELEXP.
- ND_PhonErr. A phoneme-code spelling error has occurred while a CMD_WRITE command is being executed. The Narrator device internal routines cannot understand the phoneme string.
- ND_RateErr. The requested speaking rate is out of bounds.
- ND_PitchErr. The requested speaking pitch is out of bounds.
- ND_SexErr. The Narrator_rb structure sex parameter is not set to either MALE or FEMALE.

- ND_ModeErr. The requested speaking mode is not allowed.
- ND_FreqErr. The requested sampling frequency is out of bounds.
- ND_VolErr. The requested speaking volume is out of bounds.

## USE OF FUNCTIONS

## *CloseDevice*

### Syntax of Function Call

```
CloseDevice (narrator_rb)
                     A1
```

### Purpose of Function

This function closes access to Narrator device unit 0. When CloseDevice returns, the io_Device pointer in the Narrator_rb (IOStdReq substructure) will be reset (to −1) to indicate that no task currently has the Narrator device open.

CloseDevice also decrements the Device structure lib_OpenCnt parameter and the Unit structure unit_OpenCnt parameter, thus reducing these to −1 and 0, respectively. Then, if a deferred expunge for the Narrator device sent by this or another task is pending, the Narrator device routines are expunged from RAM as soon as unit 0 is closed.

When CloseDevice returns, the current task cannot use the Narrator device until it executes another OpenDevice function call.

### Inputs to Function

| | |
|---|---|
| **narrator_rb** | A pointer to a properly initialized Narrator_rb structure |

### Discussion

CloseDevice provides a way to terminate access to a set of Narrator device unit 0 internal device routines. Because unit 0 is the only valid unit, CloseDevice always closes access to the Narrator device as a whole. Because the Narrator device is a shared access mode device, each task that uses it need not call the OpenDevice and CloseDevice functions in pairs.

## CloseLibrary

### Syntax of Function Call

```
CloseLibrary (library)
              A1
```

### Purpose of Function

This standard Exec library function closes access to the Translator library for one task in the Amiga system. When CloseLibrary returns, the Library structure lib_OpenCnt parameter will be reduced by 1 to indicate that one less task has the Translator library open.

If the lib_OpenCnt parameter is reduced to 0, and a deferred expunge called by this or another task is pending, the Translator library internal routines are expunged from RAM as soon as CloseLibrary returns in the current task.

When CloseLibrary returns, the current task cannot use the Translator library again until it executes another OpenLibrary function call.

### Inputs to Function

| | |
|---|---|
| **library** | A pointer to a properly initialized Library structure that manages the Translator library routines; the pointer variable returned by a call to the OpenLibrary function |

### Discussion

The Translator library CloseLibrary function provides a way for a task to terminate access to the internal routines of the Translator library. Each task that uses the Translator library must eventually call the CloseLibrary function; OpenLibrary and CloseLibrary function calls always occur in pairs.

## OpenDevice

### Syntax of Function Call

```
error = OpenDevice ("narrator.device", 0, narrator_rb, 0)
D0                   A0                D0  A1          D1
```

## Purpose of Function

This function opens access to Narrator device unit 0 and passes a set of CMD_WRITE command parameters to its internal routines. These parameters specify a phoneme string for the Narrator device. The Narrator device is always opened in shared access mode.

A 0 value returned in io_Error indicates that the requested opens and Audio device channel allocations succeeded. If OpenDevice is successful, it initializes the Narrator_rb structure IOStdReq substructure io_Device pointer. It also assigns a pseudo unit number to the io_Unit pointer parameter. This Unit structure and its MsgPort substructure will be used in synchronized read and write operations.

If unsuccessful, OpenDevice returns an io_Error value as follows:

- IOERR_OPENFAIL. The Narrator device could not be opened.
- ND_NoAudLib. The Audio device could not be opened because the Audio device library was not available.
- ND_NoMem. The system could not allocate enough memory to open both the Narrator and Audio devices simultaneously.
- ND_UnitErr. The unitNumber argument specified a unit other than 0.
- ND_CantAlloc. The system could not allocate the Audio device channels specified by the channel-combination array in the Narrator_rb structure ch_masks parameter.

If this is the first time the task has tried to open the Narrator device, OpenDevice will also attempt to open the Audio device. This action allocates the internal device buffers for the Audio device as defined by the Audio device's channel-combination array (see Chapter 3). The Narrator_rb structure must specify the number of entries in the channel combination array (nm_masks) and its RAM location (ch_masks) for the Audio device.

Once OpenDevice has successfully opened the Narrator and Audio devices, it initializes parameters in the Narrator_rb structure. If the calling task wants to use values other than the defaults for these parameters, it should initialize them after the OpenDevice function call returns successfully.

OpenDevice also increments the Device (Library) structure lib_OpenCnt parameter and the Unit structure unit_OpenCnt parameter, thereby keeping the Narrator device internal routines from being expunged from the system by a deferred expunge.

OpenDevice requires a properly initialized task reply-port MsgPort structure with a task signal bit allocated to that message port. The task can then be signaled when the CMD_WRITE command is executed and sent back to the task reply-port queue.

## Inputs to Function

| | |
|---|---|
| **"narrator.device"** | A pointer to a null-terminated string representing the name of the Narrator device |
| **0** | The Narrator device unit number |

| | |
|---|---|
| **narrator_rb** | A pointer to a Narrator_rb structure that represents a CMD_WRITE command |
| **0** | Indicates that the flags parameter is not used for the Narrator device |

## Preparation of the Narrator_rb Structure

Initialize the following Narrator_rb parameters:

- mn_ReplyPort. Set this to point to a MsgPort structure representing the task reply port. This message port will receive the CMD_WRITE I/O request reply from the Narrator device internal routines when the OpenDevice function call is processed.
- io_Command. Set this to CMD_WRITE.
- io_Data. Set this to point to a phoneme-string buffer whose contents the Narrator device internal routines will attempt to narrate.
- io_Length. Set this to the number of characters in the phoneme-string buffer that you want the Narrator device to speak.
- ch_masks. Set this to point to a channel combination array to be used by the Audio device internal routines.
- nm_masks. Set this to the number of entries in the Audio device channel combination array.
- mouth. Set this to 1 to tell the Narrator device to compute mouth shapes to be processed by a sequence of CMD_READ commands.

## Discussion

The OpenDevice function opens the Narrator device routines for access by a task. The Narrator device can only be opened in shared access mode. It is usually opened with a set of Narrator_rb structure parameters required to define a unit 0 CMD_WRITE command. Once this command is initiated, a series of CMD_READ commands can be dispatched to display mouth shapes consistent with the spoken syllables coming back from the Audio device internal routines.

When OpenDevice returns, the Narrator_rb structure io_Device pointer parameter will point to a Device structure that will be used to manage Narrator device unit 0; unit 0 will also have a Unit structure that defines a message-port list for all I/O requests sent to it.

The Narrator_rb structure ch_masks and nm_masks parameters must be initialized for the Audio device before OpenDevice is called. In addition, OpenDevice is always opened with several parameters initialized in the Narrator_rb structure IOStdReq substructure.

The mouth parameter of the Narrator_rb structure should also be initialized to 1 to tell the Narrator device to compute a continuous series of mouth shapes. These can then be used as input to a continuous loop of CMD_READ commands.

## *OpenLibrary*

### Syntax of Function Call

```
translatorBase = OpenLibrary ("translator.library", 0)
D0                                  A1                 D0
```

### Purpose of Function

This standard Exec function opens access to the Translator library for one task in the Amiga system. When OpenLibrary returns, the lib_OpenCnt pointer in the Library structure that manages the Translator library will be increased by 1 to indicate that one more task has the Translator library open.

OpenLibrary returns a pointer to a TranslatorBase structure in the pointer variable translatorBase, which is a global variable; all tasks in the system that want to open the Translator library must use it to open the Translator library in those tasks. The translatorBase variable must be declared as a LONG EXTERN (external) variable.

The TranslatorBase structure contains a Library structure as its first substructure. Therefore, the translatorBase pointer variable also points to a Library structure. Each task uses the same Library structure to manage the Translator library in that task.

### Inputs to Function

| | |
|---|---|
| **"translator.library"** | The name of the Translator library |
| **0** | Tells the system that the task will accept any version of the Translator library currently on disk |

### Discussion

Each task that wants to use the Translate function must open the Translator library with an OpenLibrary call. Then that task can use the Translate function to translate English strings to phoneme strings, which can then be passed on to the Narrator device internal routines and eventually to the Audio device internal routines.

The Translator library provides only one function that is directly available by a C language function call. In contrast, the Narrator and Audio device libraries also contain device commands that can be packaged into I/O request structures and dispatched by the BeginIO, DoIO, and SendIO functions.

The Translator library can be open in any number of tasks simultaneously. Each of these tasks will use the same TranslatorBase structure to manage the Translator library while it is open in that task. Only when all tasks that have opened the Translator library have also closed it can it be expunged from the system.

Note that the Translator library is a disk-resident library. Therefore, for the OpenLibrary function call to succeed, the AmigaDOS LIBS: directory must contain the translator.library file.

## *Translate*

### Syntax of Function Call

```
return_code = Translate (englishString, english_string_length,
D0                       A0             D0
                         phonemeString,phoneme_string_length)
                         A1             D1
```

### Purpose of Function

This function converts an English string into an equivalent phoneme string. The Translate function will return a 0 value in the return_code variable if no error occurs during English-to-phoneme string translation processing.

The only error that can occur is an overflow in the phoneme-string buffer; if this is about to occur, Translate stops the ongoing translation at a word boundary before the overflow takes place. The Translate function then returns a negative return_code value. Its absolute value represents the character location in the English string where the translation ceased. The calling task can then use the absolute value as an offset from the beginning of the English-string buffer when it calls the Translate function to continue with the translation where it left off. This procedure fills the phoneme-string buffer with a new set of characters that can then be passed on to the Narrator device internal routines.

### Inputs to Function

| | |
|---|---|
| **englishString** | A pointer to an English-string buffer |

| | |
|---|---|
| **english_string_length** | The number of characters in the English-string buffer, including spaces, punctuation marks, and other ASCII characters |
| **phonemeString** | A pointer to the phoneme-string buffer that the Translate function fills with phoneme characters equivalent to the ASCII characters in the English-string buffer |
| **phoneme_string_length** | The maximum expected number of characters that the Translate function will place in the phoneme-string buffer |

### Discussion

The Translate function is the only function currently in the Translator library. Its purpose is to take a task-specified English string and convert it into an equivalent phoneme string according to the rules of the Amiga's built-in Voice Synthesis library.

Normally, you use the Translate function to pass phoneme-string characters to the Narrator device and subsequently to the Audio device. The precise source of English-string characters is not important; the only requirement is that they are placed into the English-string buffer according to certain rules that allow the Translate function to work properly. These rules are beyond the scope of this volume.

The phoneme-string buffer should usually be larger than the English-string buffer; the conversion from ASCII characters to phoneme characters usually results in more phoneme characters. Therefore, to be safe, always specify the english_string_length parameter to be less than the phoneme_string_length value.

## STANDARD DEVICE COMMANDS

## *CMD_FLUSH*

### Purpose of Command

This command aborts all active and queued I/O requests. CMD_FLUSH always clears the Narrator_rb structure io_Error parameter.

### Preparation of the Narrator_rb 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 Unit structures that manage Narrator device unit 0. These two parameters can always be copied from the Narrator_rb structure initialized by the OpenDevice function call.

In addition, initialize io_Command to CMD_FLUSH, and initialize all other parameters to 0.

### Discussion

The CMD_FLUSH command flushes all currently pending and active I/O requests from the unit 0 device request queue. CMD_FLUSH will abort all CMD_WRITE and CMD_READ commands now executing and any that are queued. Because CMD_FLUSH is a very destructive command, you normally use it only if you want to restore the system to some known empty device queue state.

## CMD_READ

### Purpose of Command

This command returns the next mouth shape supplied by the Narrator device internal routines as a result of the execution of a CMD_WRITE command. The Narrator device internal routines guarantee that the mouth shape returned is different from the immediately preceding mouth shape, in accordance with the narration of the phoneme string.

Each CMD_READ command I/O request is associated with a CMD_WRITE command I/O request. Because the first entry inside Mouth_rb is a Narrator_rb substructure, each task must copy the CMD_WRITE command Narrator_rb structure into the CMD_READ command Mouth_rb structure for a series of reads before the CMD_READ command is dispatched.

If no CMD_WRITE command is in progress when the task dispatches CMD_READ, or if no CMD_WRITE command is queued in the Narrator device request queue, the CMD_READ command I/O request structure will be returned to the calling task reply-port queue with the Mouth_rb structure io_Error parameter set to ND_NoWrite. The mouth shape will not then be different from the most recently returned shape.

The task that dispatches a CMD_READ command can always look at the replied Mouth_rb structure io_Error parameter to see if the ND_NoWrite bit is set. If it is, the calling task should not dispatch any more CMD_READ commands until it first dispatches another CMD_WRITE command.

The results of command execution are as follows:

- width. This is the current width of the mouth shape measured in millimeters/3.67. Division by 3.67 is required for horizontal pixel scaling.

- height. This is the current height of the mouth shape in millimeters. There is no vertical scaling.
- shape. This represents a compressed form of the mouth shape. It is for system internal use only.
- io_Error. This is the error number returned from BeginIO, DoIO, or SendIO indicating conditions during command processing; 0 indicates the command was successful. ND_NoWrite means that CMD_READ could not execute because the Narrator device was not currently executing a CMD_WRITE command.

### Preparation of the Mouth_rb Structure

First copy the CMD_WRITE Narrator_rb structure into the CMD_READ Mouth_rb structure. Then, initialize mn_ReplyPort to point to the MsgPort structure representing the desired task reply port. Initialize the io_Device and io_Unit parameters to point to the Device and Unit structures that manage Narrator device unit 0. These two parameters can always be copied from the Narrator_rb structure initialized by the OpenDevice function call.

Also initialize io_Command to CMD_READ, and initialize io_Error, width, and height to 0.

## Discussion

The CMD_READ command handles the visual display of mouth shapes on the Amiga screen. It displays different mouth shapes consistent with the phoneme-string syllables supplied to the Narrator device. The correlation of these syllables and specific mouth shapes and parameters is predefined in the Narrator device internal routines by a built-in algorithm.

For each dispatched CMD_WRITE command, a task must dispatch a series of CMD_READ commands to produce the changing mouth shapes. The dispatching of CMD_READ commands usually takes the form of a C language loop that continues until the writing process has completed. The dispatching task looks at the replied CMD_READ Mouth_rb structure io_Error ND_NoWrite parameter bit; when it is set, the CMD_WRITE command will have completed and there is no need for additional CMD_READ commands. Therefore, in your programs, you can use the io_Error ND_NoWrite parameter bit to determine when to exit the C language CMD_READ dispatching loop.

## CMD_RESET

### Purpose of Command

CMD_RESET resets unit 0, returning it to its default configuration, aborting all pending I/O requests and restarting unit 0 if it was previously stopped by CMD_STOP.

CMD_RESET always clears the Narrator_rb structure io_Error parameter.

### Preparation of the Narrator_rb 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 Unit structures that manage Narrator device unit 0. These two parameters can always be copied from the Narrator_rb structure initialized by the OpenDevice function call.

Also initialize io_Command to CMD_RESET, and initialize all other parameters to 0.

### Discussion

CMD_RESET is a destructive command. It calls CMD_FLUSH indirectly, thereby flushing all of the queued I/O requests in the unit 0 device request queue. CMD -_RESET also calls CMD_START to start unit 0 if it was previously stopped with CMD_STOP.

## CMD_START

### Purpose of Command

If unit 0 was previously stopped by the CMD_STOP command, CMD_START starts Narrator device unit 0, including any CMD_WRITE command that was stopped in the middle of its activity or the first CMD_WRITE command at the top of the unit 0 device request queue when the CMD_STOP command was originally dispatched.

CMD_START always clears the Narrator_rb structure io_Error parameter.

### Preparation of the Narrator_rb 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 Unit structures that manage Narrator device unit 0. These two parameters can always be copied from the Narrator_rb structure initialized by the OpenDevice function call.

Also initialize io_Command to CMD_START, and initialize all other parameters to 0.

### Discussion

The CMD_START command is similar to the Ctrl-Q command used to restart halted screen output on most computers. It restarts execution of CMD_WRITE commands previously stopped by the CMD_STOP command. CMD_START will also instruct the Narrator device internal routines to process other queued I/O requests.

## CMD_STOP

### Purpose of Command

The CMD_STOP command immediately stops an executing CMD_WRITE command on Narrator device unit 0. It also prevents the Narrator device internal routines from processing any currently queued write requests. Once unit 0 is stopped by the CMD_STOP command, the system automatically queues unit 0 CMD_WRITE requests until CMD-_START restarts or CMD_RESET resets unit 0.

CMD_STOP always clears the Narrator_rb structure io_Error parameter.

### Preparation of the Narrator_rb 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 Unit structures that manage Narrator device unit 0. These two parameters can always be copied from the Narrator_rb structure initialized by the OpenDevice function call.

Also initialize io_Command to CMD_STOP, and initialize all other parameters to 0.

### Discussion

The CMD_STOP command is similar to the Ctrl-S command used for screen output on most computers. CMD_STOP stops any currently executing unit 0 CMD_WRITE command at the earliest possible opportunity. It also prevents processing of any currently queued Narrator device unit 0 CMD_WRITE or CMD_READ requests.

## CMD_WRITE

### Purpose of Command

This command performs the speech narration request. It will narrate the specified phoneme string, sending the result to the Audio device internal routines to be spoken through specified Audio device channels.

The Narrator device internal routines automatically queue CMD_WRITE I/O requests if there is already another CMD_WRITE command in progress or if the channel has been stopped by CMD_STOP. Also, if there is an associated CMD_READ I/O request in the device request queue, CMD_WRITE will remove it and return an initial mouth shape to the task that dispatched the CMD_WRITE command. If a task wants to execute a series of CMD_READ commands to display mouth shapes resulting from a specific CMD_WRITE command, the mouth parameter in the Narrator_rb structure representing that CMD_WRITE command must be initialized to 1.

The result of a CMD_WRITE is an io_Error value returned from BeginIO, DoIO, or SendIO indicating conditions during command processing. 0 indicates the command was successful. Other error values are as follows:

- ND_PhonErr. This indicates that there was an error in phoneme-string input to the CMD_WRITE command.
- ND_PitchErr. This indicates that the specified speaking pitch is out of range.
- ND_RateErr. This means that the specified speaking rate is out of range.
- ND_SexErr. This indicates that the sex has not been set.
- ND_ModeErr. This indicates that the specified speaking mode is not valid.
- ND_VolErr. This means that the specified speaking volume is out of range.
- ND_FreqErr. This indicates that the specified sampling frequency is out of range.
- io_Actual. This is the number of characters in the phoneme string that were actually processed by the current CMD_WRITE command. If the io_Error parameter in the replied Narrator_rb structure indicates a phoneme error (ND_PhonErr), io_Actual is the character position in the input phoneme string where that error occurred. The phoneme string can be corrected there and the CMD_WRITE command dispatched again starting at that character position.

## Preparation of the Narrator_rb 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 Unit structures that manage Narrator device unit 0. These two parameters can always be copied from the Narrator_rb structure initialized by the OpenDevice function call.

In addition, initialize the following command-specific parameters:

- io_Command. Set this to CMD_WRITE.
- io_Data. Set this point to the phoneme-string buffer that will be narrated and passed on to the Audio device by the Narrator device. The string could have come from the Translate function.
- io_Length. Set this to the number of characters in the phoneme string.
- ch_masks. Set this to point to a channel combination array defining the preferred allocation order of Audio device channels for the Narrator device speech.
- nm_masks. Set this to the number of entries in the Audio device channel combination array. It ranges from 1 to 16.
- mouth. Set this to 0 if no mouth shapes are to be read and to 1 if mouth shapes are to be read.
- rate. Set this to the speaking rate of the narration, from 40 to 400 words per minute. The default rate is 110.
- pitch. Set this to the pitch of the narration. It can vary from 65 to 320; the default is 110.
- mode. Set this to the speaking mode of the narration. It can be 0 for a natural expressive voice or 1 for a robotic voice; the default is 0.
- sex. This indicates the sex of the speaking voice: 0 for male or 1 for female. The default is 0.

## Discussion

CMD_WRITE is the only command in the Narrator device software system that directly narrates a phoneme string to the Audio device and eventually to the external hardware of the Amiga. Any task in the system can dispatch a continuous stream of CMD_WRITE commands to Narrator device unit 0. CMD_WRITE commands will always be queued in the unit 0 device request queue; there is no QuickIO mechanism. This means that the phoneme strings will always be written (spoken) in the order in which they were queued. However, it does not mean that once the CMD_WRITE requests are queued, the operation of the system cannot be altered. The destructive CMD_FLUSH and CMD_RESET commands (and the AbortIO function) can remove specific queued I/O requests. In addition, CMD_START and CMD_STOP can start and stop the narration of phoneme-string buffer data.

CMD_WRITE merely narrates the phoneme-string data and passes it on to the Audio device. It does not produce any mouth shapes—the CMD_READ command does that. For each CMD_WRITE command, you must use a series of CMD_READ commands. They will usually be placed in a C language loop, which will execute as long as CMD_WRITE is writing to the Audio device channels.

Note that a task should set up a separate reply-port queue for its CMD_WRITE and CMD_READ commands. It can use CreatePort, an Exec library support function, to create these two message ports. Each of these message ports can have its own assigned message-port signal bit number. In this way, each Narrator device task can monitor the progress of its CMD_WRITE and CMD_READ commands separately. There will often be many CMD_READ commands for each CMD_WRITE command; separate message ports allow a task to detect continuing CMD_READ progress while a single CMD_WRITE is executing.

# 5 The Parallel Device

## Introduction

The Parallel device allows one or more tasks to communicate with external hardware devices connected to the Amiga's parallel port. The Parallel device is disk-resident, and it must be present in the DEVS: directory if the linker is to find it during the compilation and linking process. Once the Parallel device has been loaded from disk with the first OpenDevice call, other tasks will get its routines from RAM. The most common use of the Parallel device is to drive a parallel printer connected to the Amiga's parallel port.

Parallel device commands allow any number of tasks to open and share the Parallel device routines using the Amiga's multitasking features. Each task can request this by specifying shared access mode when it first opens the Parallel device. The Parallel device also allows each task to specify up to eight characters in a character termination array that controls read and write operations between a task and an external hardware device connected to the parallel port. These characters cause read or write operations to stop when one of them is encountered in the input or output stream. This arrangement allows a task to deal with a block-oriented external hardware device, where the data blocks are separated by known ASCII characters.

## Read-Write Operations for the Parallel Device

Parallel device read-write operations are controlled by two commands: CMD_READ and CMD_WRITE. Figure 5.1 shows how a read-write operation works.

The Parallel device deals with two task-defined buffers—one for read operations and the other for write operations. The IOExtPar structure io_Data parameter serves as a pointer to each of these buffers. As Figure 5.1 indicates, the transfer of data values in and out of the buffers can always be stopped and restarted using CMD_STOP and CMD-_START. The Parallel device does not have any internal device buffers at this time, so CMD_CLEAR is an inoperative command. The current internal operation of CMD-_CLEAR is very simple: it merely executes a return from subroutine (68000 RTS) instruction. Data is transferred directly to and from the parallel-port data register without being held in an intermediate internal buffer.

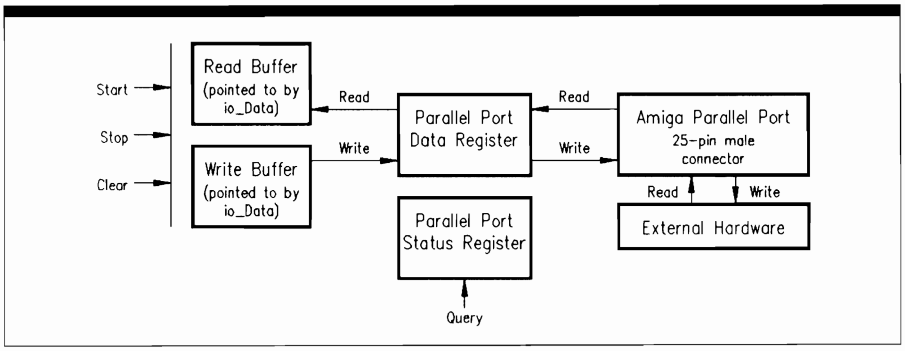

Start
Stop
Clear
Read Buffer (pointed to by io_Data)
Write Buffer (pointed to by io_Data)
Read
Write
Parallel Port Data Register
Read
Write
Amiga Parallel Port 25-pin male connector
Read
Write
External Hardware
Parallel Port Status Register
Query

**Figure 5.1:** *Read-Write Operations for the Parallel Device*

When the Parallel device is performing a read operation, all data passes through the parallel-port data register. The data then passes one byte at a time through the parallel port on the back of the Amiga and out to the external hardware connected to it. The Amiga parallel-port connector is a 25-pin connector on the right side of the back of the Amiga. Table 5.1 summarizes the pin connections. If you want to use the parallel port, you must buy a cable that is compatible with this pin arrangement.

The Parallel device internal routines maintain an up-to-date value in the parallel-port status register; this value is also the IOExtPar structure io_Status parameter. You can use the PDCMD_QUERY command to determine the value of each bit in the register whenever the Parallel device is active. Bit meanings are given in Table 5.2.

## Parallel Device Commands

The Parallel device has a total of nine commands—seven standard device commands and two device-specific commands. The only standard command not supported by the Parallel device is the CMD_UPDATE command—the Parallel device does not support any internal device buffers.

PDCMD_QUERY is a status command that inquires about the state of the system; all other commands are action commands, which change something in the system. CMD_READ and CMD_WRITE support both queued I/O and QuickIO. Seven commands execute as immediate-mode commands if successful. The only two commands that do not affect the IOExtPar structure io_Error parameter are CMD_CLEAR and PDCMD_QUERY. All other Parallel device commands return an io_Error value.

### Sending Commands to the Parallel Device

Figures 5.2(a) and (b) depict the general scheme used to send commands to the Parallel device internal routines. The lines with arrows represent the parameters you should initialize and those returned by the device internal routines.

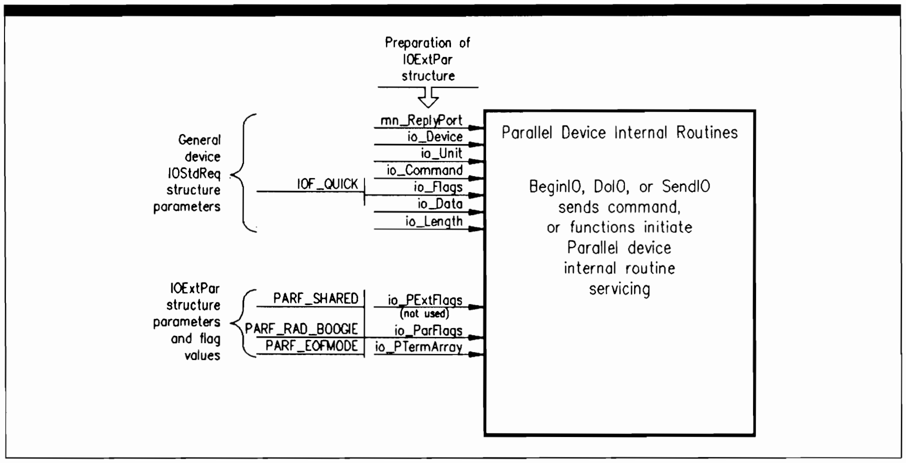

Preparation of IOExtPar structure
General device IOStdReq structure parameters
mn_ReplyPort
io_Device
io_Unit
io_Command
IOF_QUICK
io_Flags
io_Data
io_Length
IOExtPar structure parameters and flag values
PARF_SHARED
io_PExtFlags (not used)
PARF_RAD_BOOGIE
io_ParFlags
PARF_EOFMODE
io_PTermArray
Parallel Device Internal Routines
BeginIO, DoIO, or SendIO sends command, or functions initiate Parallel device internal routine servicing

**Figure 5.2(a):** *Parallel Device Command and Function Processing (Specifications)*

The Parallel device programming process consists of three phases:

1. IOExtPar structure preparation. The programmer has complete control over this phase. Here, you initialize parameters in the IOExtPar structure in preparation for

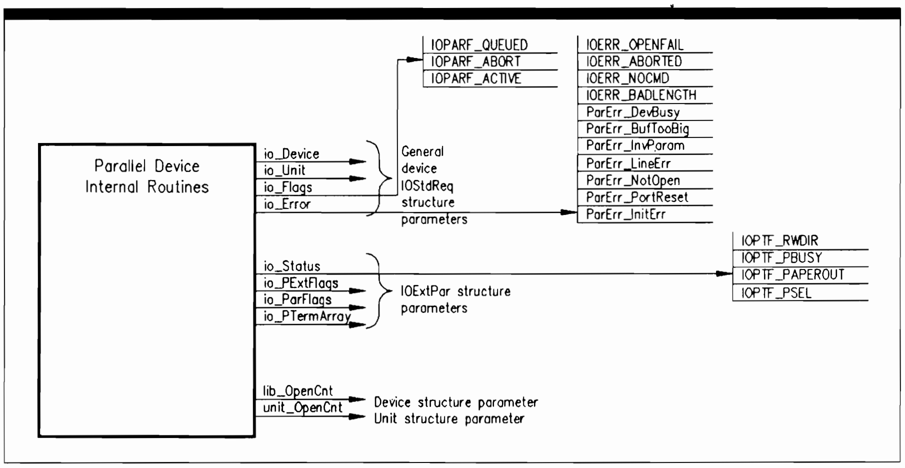


**Figure 5.2(b):** *Parallel Device Command and Function Processing (Outputs)*

**Table 5.1:** *Pin Connections for the Parallel Port*

| Pin No. | Data at Pin | Description |
|---|---|---|
| 1 | DRDY | Output data ready signal |
| 2 | D0 | 8-bit bidirectional data signal |
| 3 | D1 | 8-bit bidirectional data signal |
| 4 | D2 | 8-bit bidirectional data signal |
| 5 | D3 | 8-bit bidirectional data signal |
| 6 | D4 | 8-bit bidirectional data signal |
| 7 | D5 | 8-bit bidirectional data signal |
| 8 | D6 | 8-bit bidirectional data signal |
| 9 | D7 | 8-bit bidirectional data signal |
| 10 | ACK | Output data acknowledge signal |
| 11 | BUSY | Printer buffer full signal |
| 12 | POUT | Printer paper out signal |
| 13 | SEL | Select output signal |
| 14-22 | GND | Ground pins |
| 23 | +5 VOLTS | Voltage source (100ma max.) |
| 24 | NO | CONNECTION |
| 25 | RESET | Amiga system reset |

sending a command to the Parallel device routines. These parameters include the normal set of parameters required by most devices, as well as parameters that are specific to the IOExtPar structure; the choice of parameters depends on the specific command you plan to dispatch. These parameters, taken together, provide an information path to the data needed by the Parallel device internal routines to process the command or function.

2. Parallel device processing. The only part you play in this phase is to dispatch the command to the device using the BeginIO, DoIO, or SendIO function. Once one of these functions begins executing, control passes to the device and system internal routines. For CMD_READ and CMD_WRITE, the request can be either QuickIO or queued I/O; the other commands operate in immediate mode only.

3. Command output parameter processing. The system and the Parallel device routines have complete control over the values found in these parameters. Here, the results of command processing have been returned to the task that originally dispatched the command. If the I/O request was not QuickIO or immediate-mode execution, it was processed when it moved to the top of the Parallel device request queue and was then sent to the task reply-port queue. If the request was specified as QuickIO and it was successful, or if it was an immediate-mode command request, it was not queued in the device-unit request queue but went directly back to the task reply-port queue after device processing. The parameters still direct you to data appropriate for your task.

For most of the Parallel device commands, the outputs from device-command processing can be found in the io_Error and io_Status parameters. They only provide an indication of a processing error or the status of the Parallel device request; they do not provide an information path to the results needed by the requesting task. The CMD_READ and CMD_WRITE commands work with the io_Data parameter, which points to the RAM data areas where the task can find the returned data.

Figures 5.2 (a) and (b) also depict the parameters that play a part in Parallel device function setup and processing. The OpenDevice and CloseDevice functions both affect the Unit structure unit_OpenCnt parameter and the Device structure lib_OpenCnt parameter. OpenDevice also affects the io_Error parameter, and it can affect the values of io_PExtFlags, io_ParFlags, and io_PTermArray.

**Table 5.2:** *Parallel-Port Status Register Bits*

| Bit No. | Meaning of Bit |
|---|---|
| 0 | Printer selected (bit = 0) |
| 1 | Paper out (bit = 0) |
| 2 | Printer busy toggle (bit = 0) |
| 3 | Read (bit = 0); write (bit = 1) |
| 4–7 | Reserved |

## Structures for the Parallel Device

The Parallel device works with two structures, IOExtPar and IOPArray, as shown in Figure 5.3. Notice that the IOExtPar structure contains two substructures: an IOStdReq structure named IOPar, and an IOPArray structure named io_PTermArray. The IOExtPar structure does not contain pointers either to other structures or to data areas in RAM. The IOPArray structure contains no substructures and no pointers.

### The IOPArray Structure

The IOPArray structure is defined as follows:

```
struct IOPArray {
   ULONG PTermarray0;
   ULONG PTermarray1;
};
```

The parameters in the IOPArray structure are as follows:

- PTermArray0. This is a 4-byte parameter defining the first of a maximum of four ASCII characters—any of 00 (Ctrl-A) through 26 (Ctrl-Z)—in the character termination array. They are stored in descending ASCII order.
- PTermArray1. This is a 4-byte parameter defining the second of four ASCII characters in the character termination array. They are also defined and stored in descending ASCII order. The system checks for termination characters in these two arrays only if the io_ParFlags parameter PARF_EOFMODE bit is set.

Note that if less than eight characters are used, you should fill out the character termination array with a repeated set of the lowest valid values. For example, an ASCII array x0807060504030303 defines eight termination characters, the last three of which have an ASCII value of 03 (Ctrl-C). The character termination array is used by OpenDevice only if the io_ParFlags parameter PARF_EOFMODE bit is set. Each task can define its own character termination array and change it as needed with the PDCMD-_SETPARAMS command.

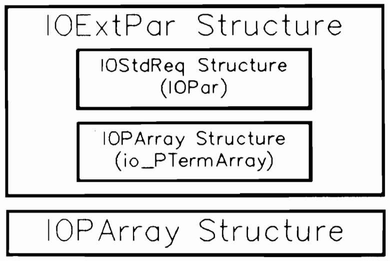

IOExtPar Structure
IOStdReq Structure (IOPar)
IOPArray Structure (io_PTermArray)
IOPArray Structure

**Figure 5.3:** *Parallel Device Structures*

## The IOExtPar Structure

The IOExtPar structure is defined as follows:

```
struct IOExtPar {
   struct IOStdReq IOPar;
   ULONG io_PExtFlags;
   UBYTE io_Status;
   UBYTE io_ParFlags;
   struct IOPArray io_PTermArray;
};
```

These are the parameters in the IOExtPar structure:

- IOPar. This is an IOStdReq structure containing an IORequest substructure, which in turn contains a Message substructure. The Message structure mn_ReplyPort parameter is used to define the task reply-port queue.
- io_PExtFlags. This is a set of additional flags. It is not used at this time and is only present for compatibility with future software releases.
- io_Status. This is the current status of the parallel port.
- io_ParFlags. This is a set of parallel flag bits.
- io_PTermArray. This is the name of the character termination array IOPArray substructure that is to be associated with the IOExtPar structure.

## Parallel-Port Flag Parameters

The IOExtPar structure flag parameters (values of io_ParFlags) are as follows:

- PARF_SHARED (bit 5). Set this bit if you want shared access mode for a set of tasks.
- PARF_RAD_BOOGIE (bit 3). This bit will enable a high-speed data transfer mode (not yet implemented).
- PARF_EOFMODE (bit 1). Set this bit if you want to enable data transfer using the IOPArray structure end-of-file (EOF) character.

The IOStdReq structure flag parameters (values of io_Flags) are as follows:

- IOPARF_QUEUED (bit 6). The system sets this bit when the CMD_READ or CMD_WRITE I/O request is queued.
- IOPARF_ABORT (bit 5). The system sets this bit when the CMD_READ or CMD_WRITE I/O request has been aborted by AbortIO or CMD_FLUSH.

- IOPARF_ACTIVE (bit 4). The system sets this bit when the CMD_READ or CMD_WRITE I/O request is currently active (being processed by the Parallel device internal routines).

The IOExtPar structure status flag parameters (values of io_Status) have the following meanings:

- IOPTF_RWDIR (bit 3). The system sets this bit when the CMD_READ or CMD_WRITE I/O request is currently being processed.
- IOPTF_PBUSY (bit 2). The system sets this bit when the printer is busy.
- IOPTF_PAPEROUT (bit 1). The system set this bit when the printer is out of paper.
- IOPTF_PSEL (bit 0). The system sets this bit when the parallel printer connected to the parallel port has been selected.

The io_Error values for the Parallel device have the following meanings:

- The IOERR_OPENFAIL, IOERR_ABORTED, IOERR_NOCMD, and IOERR _BADLENGTH bits have the same meaning for all devices (see Chapter 3).
- ParErr_DevBusy. The Parallel device is busy.
- ParErr_BufTooBig. There is a buffer error during data transfer.
- ParErr_InvParam. There is an invalid parameter in an I/O request.
- ParErr_LineErr. There is an electrical line error during a data transfer.
- ParErr_NotOpen. The Parallel device was not open when a Parallel device command request was made; the task should open the Parallel device and dispatch the command again.
- ParErr_PortReset. The system has been reset.
- ParErr_InitErr. An initialization error has occurred.

## USE OF FUNCTIONS

## *CloseDevice*

### Syntax of Function Call

**CloseDevice (ioExtPar)**
**A1**

### Purpose of Function

This function closes access to Parallel device unit 0, the only unit. The Timer device is also closed automatically in the current task. CloseDevice also decrements the Device structure lib_OpenCnt parameter and the Unit structure unit_OpenCnt parameter, reducing each of these by 1. Once these parameters are reduced to 0 and a deferred expunge sent by this or another task is pending, the Parallel device routines are expunged from RAM when CloseDevice returns.

When CloseDevice returns, the current task cannot use the Parallel device until it executes another OpenDevice function call. However, the Parallel device current parameter settings are saved for the next call to OpenDevice by this or any other task.

### Inputs to Function

| | |
|---|---|
| **ioExtPar** | A pointer to an IOExtPar structure; also a pointer to an IOStdReq structure |

### Discussion

CloseDevice terminates access to a set of device internal routines for the Parallel device. Because the Parallel device can be used in either exclusive or shared access mode, several possibilities can arise.

One task can open the Parallel device in exclusive access mode, or a series of tasks can open it in shared access mode. A combination in which one task opens the Parallel device in exclusive access mode and one or more tasks opens it in shared access mode is not possible. Each task that opens the Parallel device in exclusive access mode must always call CloseDevice before another task calls OpenDevice. Otherwise, the OpenDevice function call will return an IOERR_OPENFAIL error.

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

CloseDevice also closes the Timer device in the current task automatically. However, since the Timer device uses shared access mode, it can remain open in other tasks that have opened it either explicitly or indirectly through another device.

## *OpenDevice*

### Syntax of Function Call

```
error = OpenDevice ("parallel.device", 0,  ioExtPar, 0)
D0                  A0                 D0  A1        D1
```

## Purpose of Function

This function opens access to the internal routines of Parallel device unit 0. It also opens the Timer device if it has not already been opened in the task. The Parallel device can be opened in either exclusive or shared access mode.

Once it has successfully opened the Parallel device, OpenDevice initializes certain IOExtPar structure parameters to their most recently specified values or their default values. It also increments the Device (Library) structure lib_OpenCnt parameter and the Unit structure unit_OpenCnt parameter, thereby preventing a deferred expunge.

OpenDevice requires a properly initialized reply port with a task signal bit allocated to that port if the calling task needs to be signaled when the function call is replied. The results of command execution are as follows:

- io_Device. This points to a Device structure that will manage Parallel device unit 0 once it has been opened.
- io_Unit. This points to a Unit structure that will be used to define and manage a MsgPort structure for Parallel device unit 0. The MsgPort structure represents the device request queue.
- io_Error. A 0 here indicates that the requested open succeeded. IOERR_OPEN-FAIL indicates that the Parallel device could not be opened. If you try to open unit 0 in exclusive access mode but it has not been closed in another task, the OpenDevice call in the present task will return this error; if you try to open unit 0 in the same task twice while in exclusive access mode without first closing it, the second OpenDevice call will return this error. Also, if the specified unit number is not 0, OpenDevice returns the error value ND_UnitErr.

## Inputs to Function

| | |
|---|---|
| **"parallel.device"** | A pointer to a null-terminated string representing the name of the Parallel device |
| **0** | The only Parallel device unit number |
| **ioExtPar** | A pointer to an IOExtPar structure |
| **0** | Indicates that the Flags argument is not used for the Parallel device |

## Preparation of the IOExtPar Structure

Initialize mn_ReplyPort to point to a MsgPort structure for the task reply port. Initialize all other IOStdReq substructure parameters to 0 or copy them from an IOExtPar structure for a previous OpenDevice call.

Also initialize io_Command to 0, CMD_READ, or CMD_WRITE if the task should open the Parallel device and perform a read or write operation immediately. Initialize io_ParFlags to a combination of PARF_SHARED and PARF_EOFMODE if you want to open the Parallel device in shared access mode and initialize the IOExtPar IOPArray substructure; use a logical AND (bitwise OR) to set both of these bits.

If the CreateExtIO function is used to create the IOExtPar structure, it must typecast its returned pointer value (a pointer to an IOStdReq structure) into a pointer to an IOExtPar structure. The pointer would then also point to the IOStdReq structure, which is the first entry in the IOExtPar structure.

### Discussion

The OpenDevice function is used to open the Parallel device routines for access by a task. Once a task owns the Parallel device, it can dispatch a series of CMD_WRITE and CMD_READ commands (with BeginIO, DoIO, or SendIO) to send information back and forth between a task and any external hardware connected to the Amiga parallel port. When finished, the task should close the Parallel device.

The Parallel device can be opened in either exclusive or shared access mode; exclusive is the default. If a number of tasks open the Parallel device in shared access mode, none of those tasks have to close it before another task opens it, but all such tasks must close the Parallel device eventually.

Once the Parallel device is opened, other IOExtPar structure parameters can be initialized to define I/O request structures for reads and writes. Any parameters that are not explicitly initialized will return their previous values or the default values assigned by the Parallel device internal routines. If the calling task wants to use values other than the default values for these parameters, it should initialize them after OpenDevice returns.

Note that there is a close relationship between OpenDevice and the PDCMD_SETPARAMS command. A task will often use these two together; they both affect Parallel device parameters.

## STANDARD DEVICE COMMANDS

## *CMD_FLUSH*

### Purpose of Command

CMD_FLUSH aborts all active and queued CMD_READ and CMD_WRITE I/O requests. It is always executed as an immediate-mode command; all aborted I/O requests are replied to the task reply port with the io_Error IOERR_ABORTED bit set.

The results of command execution are found in the io_Error parameter. 0 indicates that the command was successful. ParErr_InvParam indicates that a task specified an invalid IOExtPar structure parameter in the CMD_FLUSH command. ParErr_NotOpen indicates that the Parallel device has not yet been opened in the task; the task should execute OpenDevice and dispatch CMD_FLUSH again.

### Preparation of the IOExtPar 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 Unit structures that manage Parallel device unit 0. These can always be copied from the IOExtPar structure initialized by the OpenDevice function call. Also initialize io_Command to CMD_FLUSH, and set io_Flags to 0.

### Discussion

The CMD_FLUSH command flushes all active and pending CMD_READ and CMD_WRITE I/O requests from the unit 0 device request queue. Because CMD_FLUSH is destructive, you should use it only if you want to restore the system to a known state with an empty Parallel device request queue. CMD_FLUSH does not affect the state of any task reply-port queue where previously replied CMD_READ, CMD_WRITE, or other command requests may be queued.

## *CMD_READ*

### Purpose of Command

The CMD_READ command causes a stream of characters to be read into a task-defined buffer from the Parallel device I/O register. The number of characters is specified by the IOExtPar structure io_Length parameter. If $-1$ is specified, the Parallel device will read characters until an EOF (end-of-file) character is read. The system default EOF character is 0. In addition, if the io_ParFlags PARF_EOFMODE bit is set, CMD_READ will continue to read characters until the first EOF character defined by the IOPArray structure is read.

A CMD_READ command can be terminated early if a read error occurs or if an end-of-file condition is encountered. The number of characters actually read is then stored in the IOExtPar io_Actual parameter.

CMD_READ can be treated as a synchronous or an asynchronous I/O request. If the mn_ReplyPort parameter is specified, the CMD_READ request structure is always replied to the calling task reply-port queue. The results of command execution are as follows:

- io_Actual. This is the number of characters actually read. It is set if an error occurred or if an EOF character was encountered during the reading process.
- io_Error. 0 indicates that the command was successful. ParErr_DevBusy indicates that the Parallel device was busy and could not execute the CMD_READ command when requested. ParErr_InvParam indicates that a task specified an invalid parameter in the IOExtPar structure used to define CMD_READ. ParErr_BufTooBig indicates that the read buffer defined by the io_Length parameter is too long. ParErr_LineErr indicates that a line error occurred during the read operation; this usually means a bad electrical connection between the external device and the Amiga parallel port. ParErr_NotOpen indicates that the Parallel device has not yet been opened in the task, which should execute OpenDevice and dispatch CMD_READ again.

## Preparation of the IOExtPar 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 Unit structures that manage Parallel device unit 0. These can always be copied from the IOExtPar structure initialized by the OpenDevice function call. Set io_Command to CMD_READ. Also initialize the following parameters:

- io_Flags. Set this to IOF_QUICK for QuickIO; otherwise, set it to 0.
- io_ParFlags. Set this to PARF_EOFMODE if you want CMD_READ to continue reading characters until it reaches one of the eight possible characters in the IOPArray structure.
- io_Length. Set this to the number of characters to be received from the Amiga parallel port, or set it to $-1$ to tell the task to receive characters until an EOF character is read into the task-defined read buffer. Always specify io_Length as larger than the number of characters expected under the most extreme circumstances. If io_Length is too small, the task-defined read buffer will overflow and any RAM contiguous with the end of the read buffer will be overwritten; the system may crash when it tries to access that RAM and finds information it cannot use or understand.
- io_Data. Set this to point to the task's read buffer, where characters coming in from the Amiga parallel port will be placed.

### Discussion

CMD_READ allows a task to place data into a task-defined read buffer as it comes from the hardware connected to the Amiga's parallel port. Data is transferred from the external hardware into the parallel-port data register and then into the task-defined buffer one character at a time.

Data will continue to be read until the system detects a 0 character in the data or until it detects one of a set of data-transfer termination characters defined in the IOPArray structure. A maximum of eight characters can be defined. This arrangement allows a task to tailor its read operations to each piece of external hardware. For example, if a physical device connected to the parallel port is known to put Ctrl-Z characters into its output stream to set off blocks of data, the task can define a character termination array consisting of eight Ctrl-Z characters, allowing the task to stop reading after each block of data comes in from the external device.

## *CMD_RESET*

### Purpose of Command

CMD_RESET resets unit 0 to the boot-up time state as if it had just been initialized. All Parallel device parameters are set to their default values. All pending CMD_READ and CMD_WRITE I/O requests for Parallel device unit 0 are aborted and unit 0 is restarted if it was previously stopped by CMD_STOP.

CMD_RESET is always executed as an immediate-mode command. All aborted I/O requests are replied to the task reply port with the io_Error IOERR_ABORTED bit set. The results of command execution are found in io_Error; 0 indicates that the command was successful. ParErr_InvParam indicates that an invalid IOExtPar structure parameter was specified. ParErr_NotOpen indicates that the Parallel device has not yet been opened in the task, which should execute OpenDevice and dispatch CMD_RESET again.

### Preparation of the IOExtPar 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 Unit structures that manage Parallel device unit 0. These can always be copied from the IOExtPar structure initialized by the OpenDevice function call. Also initialize io_Command to CMD_RESET, and set the io_Flags parameter to 0.

### Discussion

CMD_RESET is destructive—it calls CMD_FLUSH indirectly, thereby flushing all of the queued CMD_READ and CMD_WRITE I/O requests in the device request queue. In addition, CMD_RESET calls the CMD_START command to start unit 0 if it was previously stopped with CMD_STOP. When a task once again starts to send Parallel device I/O requests to unit 0, there will be no need to restart it. CMD_RESET also resets Parallel device parameters to their default values.

## *CMD_START*

### Purpose of Command

CMD_START restarts reads and writes to and from a channel if unit 0 was previously stopped by CMD_STOP. This is done by reactivating the parallel-port handshaking sequence. It includes any CMD_READ or CMD_WRITE command that was stopped in the middle of its activity or the first CMD_READ or CMD_WRITE request at the top of the unit 0 device request queue when CMD_STOP was dispatched.

CMD_START is always executed as an immediate-mode command; it replies to the task reply port if the mn_ReplyPort parameter is specified. The results of command execution are found in the io_Error parameter; 0 indicates that the command was successful. ParErr_InvParam indicates that a task specified an invalid IOExtPar structure parameter in the CMD_START command. ParErr_NotOpen indicates that the Parallel device has not yet been opened in the task; the task should execute OpenDevice and dispatch CMD-_START again.

### Preparation of the IOExtPar 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 Unit structures that manage Parallel device unit 0. These can always be copied from the IOExtPar structure initialized by the OpenDevice function call. Also initialize io_Command to CMD-_START, and set io_Flags to 0.

### Discussion

CMD_START starts the reading and writing of data into or out of the parallel-port data register. It is similar to the Ctrl-Q command, which restarts screen output on most computers. CMD_START restarts CMD_READ or CMD_WRITE commands previously stopped by CMD_STOP, just as Ctrl-Q restarts screen output previously stopped with Ctrl-S. CMD_START will also restart processing of queued I/O requests, just as Ctrl-Q displays additional files on the screen if the user has typed file-display commands.

## CMD_STOP

### Purpose of Command

The CMD_STOP command immediately stops a currently executing unit 0 CMD_WRITE or CMD_READ command. It also prevents the Parallel device routines from starting execution of queued CMD_WRITE I/O requests. CMD_STOP does its job by discontinuing the handshaking sequence for the parallel port. Once unit 0 is stopped by CMD_STOP, the system automatically queues CMD_READ and CMD_WRITE I/O requests dispatched to unit 0 until CMD_START restarts unit 0 or CMD_RESET resets it.

CMD_STOP is always executed as an immediate-mode command; it replies to the task reply port if the mn_ReplyPort parameter is specified. The results of command execution are found in the io_Error parameter. 0 indicates that the command was successful. ParErr_InvParam indicates that a task specified an invalid IOExtPar structure parameter in the CMD_STOP command. ParErr_NotOpen indicates that the Parallel device has not yet been opened in the task; the task should execute OpenDevice and dispatch CMD-_STOP again.

### Preparation of the IOExtPar 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 Unit structures that manage Parallel device unit 0. These can always be copied from the IOExtPar structure initialized by the OpenDevice function call. Also initialize io_Command to CMD-_STOP, and set io_Flags to 0.

### Discussion

The CMD_STOP command stops the execution of a CMD_READ or CMD_WRITE command. It is similar to the Ctrl-S command used for screen output on most computers; it stops a currently executing unit 0 CMD_WRITE command at the earliest possible opportunity. The Parallel device will then continue to queue any subsequent CMD_READ or CMD_WRITE requests.

## CMD_WRITE

### Purpose of Command

CMD_WRITE causes a stream of characters to be written from a task-defined buffer, one at a time, into the Parallel device data register. The number of characters is specified in the IOExtPar structure io_Length parameter; if −1 is specified, the Parallel device will write characters until an EOF (end-of-file) character is written. A CMD_WRITE command can be terminated early if a write error occurs or if an EOF condition is encountered; in this case, the number of characters written is stored in the IOExtPar structure io_Actual parameter.

CMD_WRITE can be treated as a synchronous or an asynchronous I/O request. It can be dispatched as QuickIO and always replies to the task reply-port queue if mn-_ReplyPort is specified.

The results of command execution are as follows:

- io_Actual. This indicates the number of characters actually written. The Parallel device internal routines will set this value if an error occurred or an EOF character was encountered during the writing process.

- io_Error. 0 indicates that the command was successful. ParErr_DevBusy indicates that the Parallel device was busy and could not execute the CMD_WRITE command when requested. ParErr_InvParam indicates that a task specified an invalid IOExtPar structure parameter to define the CMD_WRITE command. ParErr_BufTooBig indicates that the write buffer defined by the io_Data and io_Length parameters is too long. ParErr_LineErr indicates that a line error occurred during the write operation; this usually means a bad electrical connection between the external device and the Amiga parallel port. ParErr_NotOpen indicates that the Parallel device has not yet been opened in the task; the task should execute OpenDevice and dispatch CMD_WRITE again.

### Preparation of the IOExtPar 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 Unit structures that manage Parallel device unit 0. These can always be copied from the IOExtPar structure initialized by the OpenDevice function call.

Also initialize the following parameters:

- io_Command. Set this to CMD_WRITE.

- io_Flags. Set this to IOF_QUICK for QuickIO; otherwise, set it to 0.

- io_ParFlags. Set this to PARF_EOFMODE if you want CMD_WRITE to continue writing characters until it reaches one of the eight possible EOF characters in the IOPArray structure.
- io_Length. Set this to the number of characters to send to the parallel port, or set it to −1 to tell the task to write characters until an EOF character is written to the parallel-port data register from the task-defined write buffer.
- io_Data. Set this to point to the task's write buffer, where characters going out to the parallel-port data register originate.

### Discussion

CMD_WRITE allows a task to send data from task-defined buffers to the parallel-port data register and eventually out to the hardware connected to the Amiga parallel port. Data is transferred one character at a time. It will continue to be written until the system detects a 0 character in the data or until it detects a write termination character defined in the IOPArray structure. A maximum of eight characters can be defined. If the io_ParFlags PARF_EOFMODE bit is set, the system will continue writing characters until any one of these characters is detected in the output stream.

This arrangement allows a task to tailor its write operations to each piece of external hardware. For example, if a printer connected to the parallel port requires blocks of data separated by Ctrl-Z characters, the task can define a character termination array consisting of eight Ctrl-Z characters. A large task-defined buffer can then be set up with Ctrl-Z characters between blocks of data in that buffer. Then, when CMD_WRITE is executed, the task will stop writing after each block of data is written to the external device. The task can execute a number of CMD_WRITE commands, and each of these can send a new block of data out to the external device. The Ctrl-Z characters will also be written, allowing the external device to properly format its data.

## DEVICE-SPECIFIC COMMANDS

## *PDCMD_QUERY*

### Purpose of Command

The PDCMD_QUERY command allows a task to determine the current status of the Parallel device internal routines. A task can use PDCMD_QUERY to determine if the routines are currently reading or writing to a hardware device connected to the Amiga's parallel port—in particular, to a parallel printer. The io_Status parameter is kept up to date as events in the system change the status of hardware and software.

PDCMD_QUERY is always executed as an immediate-mode command; it replies to the task reply port if the mn_ReplyPort parameter is specified. The results of command execution are found in the io_Error parameter. 0 indicates that the command was successful. ParErr_InvParam indicates that a task specified an invalid IOExtPar structure parameter in the PDCMD_QUERY command. ParErr_NotOpen indicates that the Parallel device has not yet been opened in the task; the task should execute OpenDevice and dispatch PDCMD_QUERY again.

In addition, each bit in the IOExtPar structure io_Status parameter returns the following values:

- IOPTF_PSEL (bit 0). A 0 value indicates that a printer connected to the Amiga parallel port has been selected by the device routines and is now on-line to receive data specified by CMD_WRITE.
- IOPTF_PAPEROUT (bit 1). A 0 value indicates that the printer connected to the parallel port is out of paper; the task should tell the user to add paper. For example, an Intuition task could display a Requester alert.
- IOPTF_PBUSY (bit 2). A 0 value indicates that the printer is currently busy. If a task detects this when it attempts to dispatch a CMD_WRITE command to a printer, it should keep testing until the bit value changes to 1; it should then resubmit CMD_WRITE.
- IOPTF_RWDIR (bit 3). A 0 value indicates that the Parallel device routines are executing a CMD_READ command; 1 indicates that they are executing a CMD_WRITE command.

Bits 4–7 are reserved for future enhancements to the Parallel device.

## Preparation of the IOAudio 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 Unit structures that manage Parallel device unit 0. These can always be copied from the IOExtPar structure initialized by the OpenDevice function call. Also initialize io_Command to PDCMD_QUERY, and set the io_Flags parameter to 0.

## Discussion

The PDCMD_QUERY command is included in the Parallel device software system so that a task can monitor the activity of the Parallel device and the external hardware connected to the parallel port. The task can then decide what to do next based on the current activity.

PDCMD_QUERY is usually used to monitor a printer attached to the parallel port and driven by CMD_WRITE commands coming from a task. The task needs to know about the printer's external conditions—if the printer is currently selected, if it is busy printing, or if it is out of paper. The replied IOExtPar structure io_Status parameter determines the current status of the printer. In addition, the task must know whether the Parallel device is reading or writing to the external device. This information is provided by the io_Status IOPTF_RWDIR parameter (bit 3).

## PDCMD_SETPARAMS

### Purpose of Command

PDCMD_SETPARAMS allows a task to set the Parallel device parameters. It will only be successful if there are no active or queued CMD_READ and CMD_WRITE command requests already dispatched. A task can use the PDCMD_SETPARAMS command to set the IOExtPar structure parameters (io_PExtFlags, io_ParFlags, and io_PTerm-Array) before dispatching a CMD_READ or CMD_WRITE request. These parameters can be initialized or reinitialized with the command.

PDCMD_SETPARAMS executes as an immediate-mode command and replies to the task reply port. The results of command execution are found in io_Error; 0 indicates that the command was successful. ParErr_InvParam indicates that a task specified an invalid IOExtPar structure parameter in the PDCMD_SETPARAMS command. ParErr_Not-Open indicates that the Parallel device has not yet been opened in the task, which should execute OpenDevice and dispatch PDCMD_SETPARAMS again.

### Preparation of the IOExtPar 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 Unit structures that manage Parallel device unit 0. These can always be copied from the IOExtPar structure initialized by the OpenDevice function call.

Also initialize the following command-specific parameters:

- io_Command. Set this to PDCMD_SETPARAMS.
- io_Flags. Set this to 0.
- io_PExtFlags. Set this to 0; it is provided for future enhancements and is not used with Release 1.2 software.
- io_ParFlags. Set this to PARF_SHARED to open the Parallel device in shared access mode in a subsequent OpenDevice call; exclusive access mode is the default.

Set io_ParFlags to PARF_EOFMODE if you want to use a set of task-defined EOF characters to control end-of-file conditions for subsequent CMD_READ and CMD_WRITE requests. PARF_RAD_BOOGIE accesses the Parallel device in high-speed mode; it is not implemented at present. Note that the io_ParFlags parameter is always initialized by the new OpenDevice function call to default values or to the parameter settings made with previous PDCMD_SETPARAMS commands.

- io_PTermArray. Set this to point to a descending ASCII order 8-byte array defining a character termination array for the task. This parameter is initialized by a new OpenDevice call to reflect the current state and configuration of the Parallel device routines only if the io_ParFlags PARF_EOFMODE bit is set in the OpenDevice IOExtPar structure.

## Discussion

The PDCMD_SETPARAMS command allows a task to change the Parallel device parameters used by subsequent CMD_READ and CMD_WRITE command requests. It is usually used to set Parallel device parameters before an OpenDevice call, or to change them after an OpenDevice call but before the next CMD_READ or CMD_WRITE command is dispatched. PDCMD_SETPARAMS and the OpenDevice function interact throughout a task's execution.

PDCMD_SETPARAMS is particularly important in defining characters for terminating a CMD_READ or CMD_WRITE request. It allows a task to change the current EOF character definition in order to terminate subsequent CMD_READ or CMD_WRITE commands. If a task is dealing with a number of hardware devices with different data characteristics (for example, different character-block termination characters), it can change the current character termination array for the next CMD_READ or CMD_WRITE command in order to relate to the next hardware device with which it wants to communicate.

You can also define a separate task for each device attached to the Amiga's parallel port. You would then specify separate character termination arrays so that the tasks could operate with the device characteristics and block definitions.

# 6

# The Serial Device

## Introduction

The Serial device allows tasks to communicate with external hardware devices—most commonly a serial modem or a serial printer—connected to the Amiga's serial port. The Serial device is disk-resident and it must be present in the DEVS: directory if the linker is to find it during the compilation and linking process. Once the Serial device has been loaded from disk with the first OpenDevice call, other tasks will get its routines from RAM.

The Serial device is similar to the Parallel device. However, there are some important differences between them: in particular, the Serial device has an internal device buffer that it uses for CMD_READ execution. This buffer is called the Serial device internal read buffer.

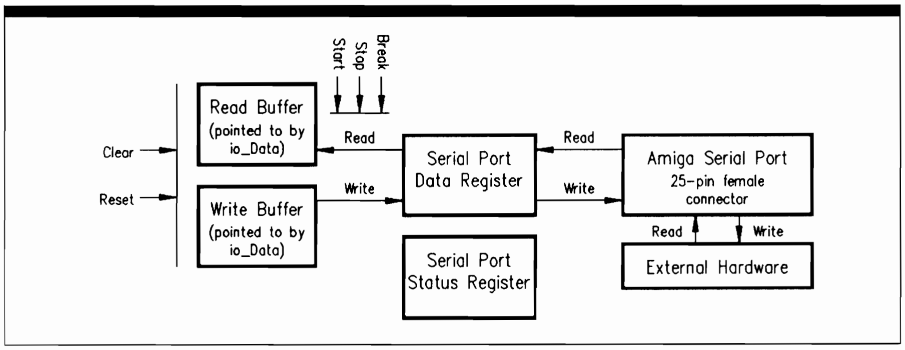
Serial device commands allow any number of tasks to open and share the Serial device internal routines using the multitasking features of the Amiga. Each task can request this by specifying shared access mode when it first opens the Serial device.

The Serial device allows each task to specify up to eight characters in a character termination array, which controls read and write operations between a task and any hardware device connected to the serial port. These characters cause the read or write operations to stop when one of them is encountered in the input or output stream. Among other things, this arrangement allows a task to deal with a block-oriented external hardware device, where the data blocks are separated by known ASCII characters.

## Read-Write Operations for the Serial Device

Serial device read-write operations are controlled by two commands: CMD_READ and CMD_WRITE. Figure 6.1 shows how read-write operations work.

The Serial device works with two task-defined buffers—one for read operations and one for write operations. The IOExtSer structure io_Data pointer serves as a pointer to each of these buffers. In addition, CMD_READ uses a Serial device internal read buffer with a default size of 512 bytes. (The CMD_WRITE command does not use an internal device buffer.) The data values in the device internal read buffer can be altered using



Start
Stop
Break
Read Buffer (pointed to by io_Data)
Write Buffer (pointed to by io_Data)
Clear
Reset
Read
Write
Serial Port Data Register
Serial Port Status Register
Read
Write
Amiga Serial Port 25-pin female connector
Read
Write
External Hardware

**Figure 6.1:** *Read-Write Operations for the Serial Device*

CMD_RESET or CMD_CLEAR; CMD_START and CMD_STOP can be used to affect data being placed into the task-defined buffers.

When the Serial device is executing CMD_READ, all data passes through the serial-port data register on its way to the Serial device read buffer. The data comes from the external hardware device attached to the Amiga serial port. When the Serial device is executing CMD_WRITE, all data once again passes through the Serial device data register and then passes one byte at a time through the serial port on the back of the Amiga and out to the external hardware connected to it.

The Amiga serial-port connector is a 25-pin female connector near the middle of the back of the Amiga. Table 6.1 summarizes the pin connections. If you want to use the serial port, you must buy a cable that is compatible with this pin arrangement.

The Serial device software maintains a status value in the serial-port status register; this value is also placed in the IOExtSer structure io_Status register. You can use the PDCMD_QUERY command to determine the value of each bit in the status register whenever the Serial device is active. Bit meanings are summarized in Table 6.2.

**Table 6.1:** *Pin Connections for the Serial Port*

| Pin No. | Data at Pin | Description |
|---|---|---|
| 1 | FGND | Frame ground |
| 2 | TXD | Transmit data |
| 3 | RXD | Receive data |
| 4 | RTS | Request to send |
| 5 | CTS | Clear to send |
| 6 | DSR | Data set ready |
| 7 | GND | System ground |
| 8 | CD | Carrier detect |
| 9–13 | — | Not used |
| 14 | −5 VOLTS | Power (50ma maximum) |
| 15 | AUDO | Audio output |
| 16 | AUDI | Audio input |
| 17 | EB | Buffer port clock (716KHz) |
| 18 | INT2 | Interrupt Line: Amiga Level 2 |
| 19 | — | Not used |
| 20 | DTR | Data terminal ready |
| 21 | +5 VOLTS | Power (100ma maximum) |
| 22 | — | Not used |
| 23 | +12 VOLTS | Power (50ma maximum) |
| 24 | C2 | 3.58MHz clock |
| 25 | RESB | Buffered system reset |

## Serial Device Commands

You can use ten commands to program the Serial device—seven are standard and three are device-specific. SDCMD_QUERY determines the status of the system; all other commands are action commands; they change something in the system.

### Sending Commands to the Serial Device

Figures 6.2(a) and (b) show the general scheme used to dispatch commands to the Serial device routines and to call functions in the Serial device software system. The lines with arrows represent the parameters you should initialize, as well as those returned by the Serial device internal routines.

The Serial device programming process consists of three phases:

1. IOExtPar structure preparation. Here, you initialize parameters in the IOExtPar structure in preparation for sending a command to the Serial device routines. These parameters include the usual set of parameters required by most devices, as well as parameters that are specific to the IOExtPar structure and used only with the SDCMD_SETPARAMS command. Flag parameters are also indicated in the figure; a task's choice of flag parameter bits depends on how it wants to use a specific command or function. These parameters, taken together, provide an information path to the data needed by the Serial device internal routines to process the command or function.

2. Serial device processing. The only part you play in this phase is to dispatch the command to the device using the BeginIO, DoIO, or SendIO function. Control then passes to the device and system internal routines.

**Table 6.2:** *Serial-Port Status Register Bits*

| Bit No. | Meaning of Bit |
|---|---|
| 0–2 | Reserved |
| 3 | Data set ready (bit = 0) |
| 4 | Clear to send (bit = 0) |
| 5 | Carrier detect (bit = 0) |
| 6 | Ready to send (bit = 0) |
| 7 | Data Terminal ready (bit = 0) |
| 8 | Read buffer overflow (bit = 1) |
| 9 | Break signal sent (bit = 1) |
| 10 | Break signal received (bit = 1) |
| 11 | Transmit XOFF (bit = 1) |
| 12 | Received XOFF (bit = 1) |
| 13–15 | Reserved |

3. Command output parameter processing. The system and the Serial device internal routines have complete control over this phase. Here, the results of command processing have been returned to the task that originally issued the function or command. If the I/O request was not QuickIO, it was processed when it moved to the top of the Serial device-unit request queue and was sent to the task reply-port queue. If the request was specified as QuickIO and was successful, or if it was an immediate-mode command request, it was not queued in the device-unit request queue but went directly to the device internal routines; the replied IOExtSer structure parameters still direct you to data appropriate for your task.

Outputs from device-command processing can usually be found in the io_Error, io_Actual, and io_Status parameters. These parameters provide an indication of a processing error, the status of the Serial device request, and the number of bytes transferred; they do not provide an information path to the results needed by the requesting task. The situation is different for SDCMD_SETPARAMS; see the SDCMD_SETPARAMS discussion in this chapter for details.

Figures 6.2(a) and (b) also depict the parameters that play a part in Serial device function setup and processing. The OpenDevice and CloseDevice functions both affect the Unit structure unit_OpenCnt and Device structure lib_OpenCnt parameters. OpenDevice also affects the io_Error parameter and can affect other IOExtSer structure parameters.

## Structures for the Serial Device

The Serial device works with two structures: the IOExtSer structure and the IOTArray structure, as shown in Figure 6.3. Notice that the IOExtSer structure contains two substructures: an IOStdReq structure named IOSer, and an IOTArray structure named io_TermArray. The IOExtSer structure does not contain any pointers either to other structures or to data areas in RAM, but its IOStdReq substructure does contain pointer parameters—in particular, the io_Data read or write buffer pointer. The IOTArray structure does not contain substructures or pointers to RAM data areas.

The IOExtSer structure contains the Serial device parameters required to characterize CMD_READ and CMD_WRITE commands dispatched to the Serial device routines. The IOTArray structure contains the set of eight characters for the character termination array used to control read and write operations.

### The IOTArray Structure

The IOTArray structure is defined as follows:

```
struct IOTArray {
   ULONG TermArray0;
   ULONG TermArray1;
};
```

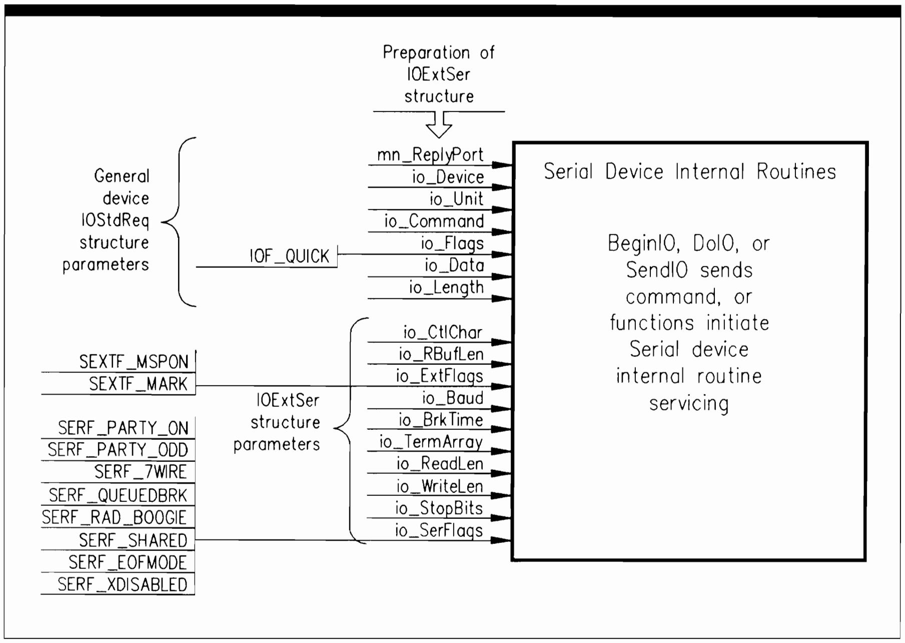


**Figure 6.2(a):** *Serial Device Command and Function Processing (Specifications)*

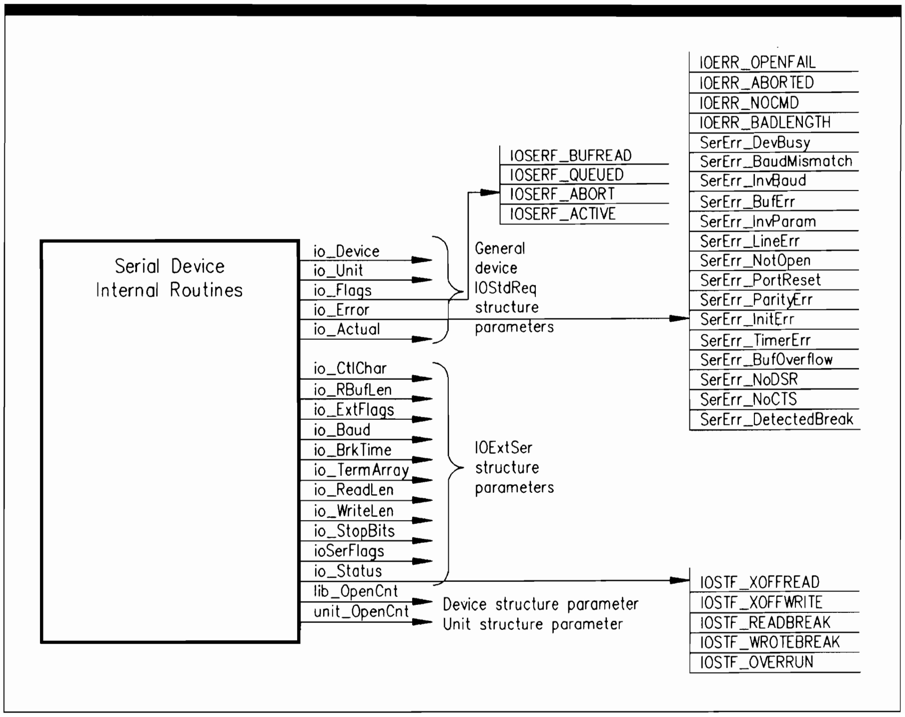


**Figure 6.2(b):** *Serial Device Command and Function Processing (Outputs)*

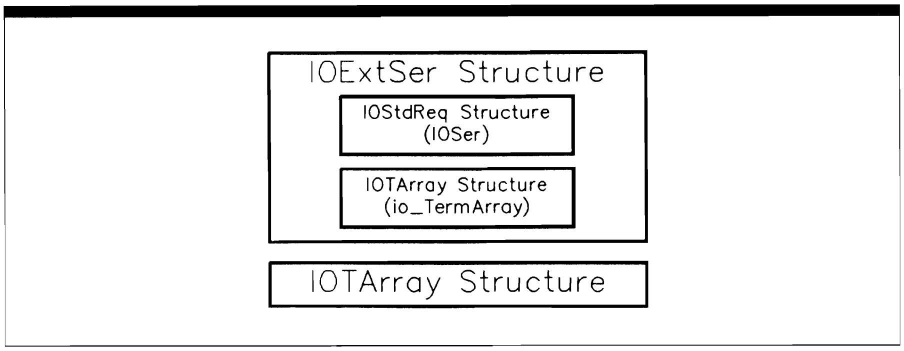


**Figure 6.3:** *Serial Device Structures*

The parameters in the IOTArray structure are as follows:

- TermArray0. This contains the first four of a total of eight hexadecimal characters in the character termination array; this array is stored in descending ASCII order.
- TermArray1. This contains the second four of the eight hexadecimal characters in the character termination array; it too is stored in descending ASCII order. The system checks for termination characters only if the IOExtSer structure io_SerFlags SERF_EOFMODE bit is set.

## The IOExtSer Structure

The IOExtSer structure is defined as follows:

```
struct IOExtSer {
    struct IOStdReq IOSer;
    ULONG io_CtlChar;
    ULONG io_RBufLen;
    ULONG io_ExtFlags;
    ULONG io_Baud;
    ULONG io_BrkTime;
    struct IOTArray io_TermArray;
    UBYTE io_ReadLen;
    UBYTE io_WriteLen;
    UBYTE io_StopBits;
    UBYTE io_SerFlags;
    UWORD io_Status;
};
```

The parameters in the IOExtSer structure are as follows:

- IOSer. This IOStdReq substructure contains the usual I/O request parameters. The mn_ReplyPort parameter specifies the task reply-port queue where the I/O request should be replied when the device internal routines are finished processing it.

- io_CtlChar. This is a set of four control characters: XON, XOFF, INQ, and ACQ. See the CMD_READ and CMD_WRITE discussions in this chapter for information on this parameter.
- io_RBufLen. This is the length (in bytes) of the Serial device internal read buffer. The default value is 512 bytes.
- io_ExtFlags. This is a set of additional flag parameter bits; it is not used in Release 1.0 or 1.1. For Release 1.2, set the SEXTF_MSPON bit if you want the Serial device to use mark-space parity instead of odd-even parity for read and write operations. Set the SEXTF_MARK bit if you want the Serial device to use a mark character when mark-space parity is selected with the SEXTF_MSPON bit.
- io_Baud. This is the requested baud rate, a number from 110 to 29,200.
- io_BrkTime. This is the duration of the break signal in microseconds; the default value is 250,000 microseconds.
- io_TermArray. This is the name of the character termination array for the IOExtSer structure.
- io_ReadLen. This is the number of bits in each character processed by the CMD_READ command.
- io_WriteLen. This is the number of bits in each character processed by the CMD_WRITE command.
- io_StopBits. This is the number of stop bits associated with every character that is read by the CMD_READ command.
- io_SerFlags. This is a set of serial flag bits defined in the following section.
- io_Status. This is the current status of the serial port.

## Serial-Port Flag Parameters

The values of the Serial device flag parameter bits (io_SerFlags) are as follows:

- SERF_XDISABLED (bit 7). Set this bit if you want the XON-XOFF protocol feature to be disabled. This protocol uses the ASCII toggle-code characters DC1 (device control 1, Ctrl-Q) and DC3 (device control 3, Ctrl-S) to start and stop data transfer.
- SERF_EOFMODE (bit 6). Set this bit if you want to enable the EOF mode, which allows a task to start and stop a data transfer using other characters in addition to the Ctrl-Q–Ctrl-S combination.
- SERF_SHARED (bit 5). Set this bit if you want shared access for the Serial device internal routines.

- SERF_RAD_BOOGIE (bit 4). Set this bit if you want to enable the high-speed data transfer mode. This is most often used to send data at high speeds through an MIDI (musical instrument digital interface).
- SERF_QUEUEDBRK (bit 3). Set this bit if you want to queue SDCMD_BREAK commands.
- SERF_7WIRE (bit 2). Set this bit if you want to use the 7-WIRE RS-232 protocol for CMD_READ and CMD_WRITE operations.
- SERF_PARTY_ODD (bit 1). Set this bit if you want to use odd parity to check the data transfer during CMD_READ or CMD_WRITE execution.
- SERF_PARTY_ON (bit 0). Set this bit if you want to use a parity check on the data transfer during CMD_READ or CMD_WRITE execution.

The values of the Serial device IOStdReq structure flag parameter bits (io_Flags) are as follows:

- IOSERF_BUFREAD (bit 7). The system sets this bit when the Serial device is reading from its internal read buffer.
- IOSERF_QUEUED (bit 6). The system sets this bit when the I/O request is treated as queued I/O rather than QuickIO.
- IOSERF_ABORT (bit 5). The system sets this bit when the I/O request has been aborted by AbortIO, CMD_RESET, or CMD_FLUSH.
- IOSERF_ACTIVE (bit 4). The system sets this bit when the I/O request is active or queued.

The values of the Serial device IOExtSer structure flag parameters bits (io_Status) are as follows:

- IOSTF_XOFFREAD (bit 4). The system sets this bit when the I/O request has just read an XOFF character.
- IOSTF_XOFFWRITE (bit 3). The system sets this bit when the I/O request has just transmitted an XOFF character.
- IOSTF_READBREAK (bit 2). The system sets this bit when the current task is receiving a break-signal input bit; a break signal is used to stop data transfer for a specified time period.
- IOSTF_WROTEBREAK (bit 1). The system sets this bit when the current task is transmitting a break-signal output bit.

- IOSTF_OVERRUN (bit 0). The system sets this bit when the I/O request has caused a Serial device input-buffer overrun.

The values for the Serial device IOStdReq io_Error parameters have the following meanings:

- IOERR_OPENFAIL, IOERR_ABORTED, IOERR_NCMD, and IOERR_BAD-LENGTH have the same meanings for the Serial device as they do for other devices (see Chapter 3).
- SerErr_DevBusy. The Serial device internal routines are busy.
- SerErr_BaudMismatch. There is a mismatch between the requested and actual baud rates.
- SerErr_InvBaud. The I/O request asked for an invalid baud rate.
- SerErr_BufErr. There was a buffer error during data transfer.
- SerErr_InvParam. There is an invalid I/O request parameter.
- SerErr_LineErr. There was a line error during a data transfer; this is usually caused by an electrical problem on the data transfer lines or connections.
- SerErr_NotOpen. The Serial device was not open when a request was made.
- SerErr_PortReset. The Serial device software system has been reset with the CMD_RESET command.
- SerErr_ParityErr. A parity error has occurred during data transfer.
- SerErr_InitErr. An initialization error occurred when the Serial device software system was initialized following a CMD_RESET command.
- SerErr_TimerErr. A timing error occurred during data transfer.
- SerErr_BufOverflow. A buffer overflow occurred during data transfer.
- SerErr_NoDSR. There was no DSR (data set ready) bit, which is required to establish the proper protocol for the data transfer.
- SerErr_NoCTS. There was no CTS (clear to send) bit, which is required to establish the proper protocol for the data transfer.
- SerErr_DetectedBreak. A break signal was detected during data transfer, thereby halting it.

## USE OF FUNCTIONS

## CloseDevice

### Syntax of Function Call

**CloseDevice (ioExtSer)**
**A1**

### Purpose of Function

This function closes access to unit 0, the only unit of the Serial device; the Timer device is also closed automatically in the current task and the Serial device internal CMD_READ buffer is freed. CloseDevice also decrements the Device structure lib_OpenCnt parameter and the Unit structure unit_OpenCnt parameter by 1. Once these parameters are reduced to 0, a deferred expunge invoked by this or another task will remove the Serial device structures from RAM when CloseDevice returns.

When CloseDevice returns, the current task cannot use the Serial device until it executes another OpenDevice function call. However, the Serial device internal parameter settings are saved for the next call to OpenDevice by this or any other task.

### Inputs to Function

| | |
|---|---|
| **ioExtSer** | A pointer to an IOExtSer structure; also a pointer to an IOStdReq structure |

### Discussion

CloseDevice terminates access to a set of device internal routines for Serial device unit 0. Because the Serial device can be used in either exclusive or shared access mode, several possibilities can arise. One task can open the Serial device in exclusive access mode, or a series of tasks can open it in shared access mode. (A situation in which one task opens the Serial device in exclusive access mode and one or more tasks opens it in shared access mode is not possible.) Each task that opens the Serial device in exclusive access mode must always call CloseDevice before another task calls OpenDevice to open the Serial device. Otherwise, the OpenDevice function call will return an io_Error IOERR_OPENFAIL value when the second task tries to open the Serial device.

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

CloseDevice also closes the Timer device in the current task automatically and frees the Serial device internal read buffer for other uses. Since the Timer device uses shared access mode, it can remain open in other tasks that have opened it either explicitly or indirectly through other devices.

## *OpenDevice*

### Syntax of Function Call

```
error = OpenDevice ("serial.device", 0,  ioExtSer, 0)
D0                  A0               D0  A1       D1
```

### Purpose of Function

This function opens access to the internal routines of Serial device unit 0. It also opens the Timer device if it has not already been opened in the task. The Serial device can be opened in either exclusive or shared access mode.

Once it has successfully opened the Serial device, OpenDevice initializes certain IOExtSer structure parameters to their most recently specified values or their default values. OpenDevice also increments the Device (Library) structure lib_OpenCnt parameter and the Unit structure unit_OpenCnt parameter, thereby preventing a deferred expunge.

OpenDevice requires a properly initialized reply port with a task signal bit allocated to that port if the calling task needs to be signaled when the OpenDevice function call is replied. The results of function execution are as follows:

- io_Device. This points to a Device structure that manages Serial device unit 0 once it has been opened.

- io_Unit. This points to a Unit structure that will be used to define and manage a MsgPort structure for Serial device unit 0. The MsgPort structure represents the unit 0 device request queue.

- io_Error. 0 indicates that the requested open succeeded. IOERR_OPENFAIL indicates that the Serial device could not be opened. If you try to open unit 0 in exclusive access mode but it has not been closed in another task, the OpenDevice call in the current task will return this error; also, if you try to open unit 0 in the same task twice using exclusive access mode without first closing it, the second OpenDevice call will return this error.

## Inputs to Function

| | |
|---|---|
| **"serial.device"** | A pointer to a null-terminated string representing the name of the Serial device |
| **0** | The only Serial device unit number |
| **ioExtSer** | A pointer to an IOExtSer structure |
| **0** | Indicates that the flags argument is ignored by Open-Device |

## Preparation of the IOExtSer Structure

Initialize mn_ReplyPort to point to a MsgPort structure for the task reply port. Initialize all other parameters in the IOStdReq substructure to 0 or copy them from a previous OpenDevice call.

Also initialize io_Command to 0, CMD_READ, or CMD_WRITE if the task should open the Serial device and then execute a read or write operation. Initialize io_SerFlags to a bitwise combination of SERF_SHARED, SERF_EOFMODE, and 7-WIRE (DTR/DSR, RTS/CTS). Do this if you want to open the Serial device in shared access mode, initialize the IOExtSer IOTArray substructure, and use the 7-WIRE (DTR/DSR, RTS/CTS) handshaking mode (RS-232C CTS/RTS protocol). You can use a logical AND (bitwise OR) to set one, two, or all three of these bits.

If the CreateExtIO function is used to create the IOExtSer structure, it must typecast its returned pointer value into a pointer to an IOExtSer structure. Then the pointer will also point to the IOStdReq structure, which is the first entry in the IOExtSer structure. (See the AbortIO discussion in Chapter 2.)

## Discussion

The OpenDevice function is used to open the Serial device routines for access by a task. Once the task owns the Serial device, it can dispatch a series of CMD_WRITE and CMD_READ commands (with BeginIO, DoIO, or SendIO) to send information back and forth between the task and any external hardware connected to the Amiga serial port. Once a task finishes all of its Serial device writing and reading, it should close the Serial device.

The Serial device can be opened in either exclusive or shared access mode; exclusive is the default. Every task that opens the Serial device in exclusive access mode must close it before another task can open it. In shared access mode, all tasks that open the Serial device must close it eventually.

All IOExtSer structure parameters that are not explicitly initialized when a task calls OpenDevice will retain their previous task-defined values or use the default values assigned by the Serial device internal routines. If the calling task wants to use other values for these parameters, it should initialize them after OpenDevice returns and the Serial device is opened.

There is a close relationship between OpenDevice and the SDCMD_SETPARAMS command. A task will often use these two together; they both affect the Serial device parameters.

## STANDARD DEVICE COMMANDS

## CMD_CLEAR

### Purpose of Command

CMD_CLEAR is designed to clear internal device buffers. The Serial device has only the internal read buffer, which CMD_CLEAR clears.

CMD_CLEAR is always an immediate-mode command. The results of command execution are found in io_Error. 0 indicates that the command was successful. SerErr_InvParam indicates that a task specified an invalid parameter in the IOExtSer structure used to define the CMD_CLEAR command. SerErr_NotOpen indicates that the Serial device has not yet been opened in the task, which should execute OpenDevice and CMD_CLEAR again.

### Preparation of the IOExtSer 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 Unit structures that manage Serial device unit 0. These can always be copied from the IOExtSer structure initialized by the OpenDevice function call. Also initialize io_Command to CMD_CLEAR, and set io_Flags to 0.

### Discussion

CMD_CLEAR directly affects the Serial device internal read buffer. (The Serial device does not use an internal write buffer.) It clears the buffer and resets the internal buffer pointer to point to the first byte in that buffer.

## CMD_FLUSH

### Purpose of Command

CMD_FLUSH aborts all queued I/O requests in the Serial device request queue. Actively executing CMD_READ and CMD_WRITE I/O requests are not affected.

CMD_FLUSH is always executed as an immediate-mode command. All aborted I/O requests are replied to the task reply-port queue with the io_Error IOERR_ABORTED bit set.

The results of command execution are found in the io_Error parameter. 0 indicates that the command was successful. SerErr_InvParam indicates that a task specified an invalid parameter in the IOExtSer structure used to define the CMD_FLUSH command. SerErr_NotOpen indicates that the Serial device has not yet been opened in the task, which should execute OpenDevice and CMD_FLUSH again.

### Preparation of the IOExtSer 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 Unit structures that manage Serial device unit 0. These can always be copied from the IOExtSer structure initialized by the OpenDevice function call. Also initialize io_Command to CMD_FLUSH, and set io_Flags to 0.

### Discussion

The CMD_FLUSH command flushes all queued I/O requests from the unit 0 device request queue. It is always an immediate-mode command. Because CMD_FLUSH is destructive, you should use it only if you want to restore the system to some known state with an empty Serial device request queue. CMD_FLUSH does not affect the state of any task reply-port queue where previously replied CMD_READ and CMD_WRITE I/O requests (and others) may be queued.

## CMD_READ

### Purpose of Command

The CMD_READ command causes a stream of characters to be read into the Serial device internal read buffer. This data passes through the Serial device data register on its way to an input buffer. When the task needs the data, it will be transferred from the Serial device internal read buffer to a task-defined buffer.

The number of characters to read is specified in the IOExtSer structure io_Length parameter. If − 1 is specified, the Serial device will read characters until an EOF character is read. (The system default EOF character is 0.) In addition, if the io_SerFlags SERF-_EOFMODE bit is set, CMD_READ will continue to read characters until the first EOF character defined by the IOTArray structure is read.

A CMD_READ command can be terminated early if a read error occurs or if an EOF character is encountered. The number of characters actually read is then stored in the IOExtSer structure io_Actual parameter.

The results of command execution are found in io_Actual and io_Error. The io_Actual parameter indicates the number of characters actually read. The Serial device internal routines will set this value if an error or an EOF character specified by the IOTArray structure was encountered during the reading process. This count does not include the CMD_READ command termination character. A 0 value in io_Error indicates that the command was successful; other io_Error values have the following meanings:

- SerErr_DevBusy. The Serial device internal routines were busy and could not execute CMD_READ as requested.
- SerErr_InvParam. A task specified an invalid parameter in the IOExtSer structure used to define CMD_READ.
- SerErr_LineErr. There was a line error during the read operation; this is usually a bad electrical connection between the external hardware device and the Amiga serial port.
- SerErr_NotOpen. The Serial device has not yet been opened in the task; the task should execute OpenDevice and CMD_READ again.
- SerErr_BufOverflow. The task-defined read buffer has overflowed; the task should change the buffer specifications and dispatch CMD_READ again.
- SerErr_BaudMismatch. The current task-specified baud rate and the external hardware-device baud rate do not match; the task should change the baud rate and dispatch CMD_READ again.
- SerErr_InvBaud. The specified baud rate is invalid (usually out of range); the task should change it and dispatch CMD_READ again.
- SerErr_BufErr. A Serial device internal read buffer error occurred during data transfer; the task should determine what caused the error and dispatch CMD_READ again.
- SerErr_ParityErr. A parity error occurred during data transfer; the task should dispatch CMD_READ again.
- SerErr_TimerErr. A timing error occurred during data transfer; the task should dispatch CMD_READ again.
- SerErr_BufOverflow. A buffer overflowed during data transfer; the task should change buffer specifications and dispatch CMD_READ again.
- SerErr_NoDSR. No DSR (data set ready) signal was sent during data transfer; the task should determine why and dispatch CMD_READ again.
- SerErr_NoCTS. No CTS (clear to send) signal was sent during data transfer; the task should determine why and dispatch CMD_READ again.

- SerErr_DetectedBreak. The system detected a queued or immediate break signal during data transfer; the task should eliminate the break signal and dispatch CMD_READ again.

## Preparation of the IOExtSer 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 Unit structures that manage Serial device unit 0. These can always be copied from the IOExtSer structure initialized by the OpenDevice function call. Set io_Command to CMD_READ.

Also initialize the following parameters:

- io_Flags. Set this to IOF_QUICK for QuickIO; otherwise, set it to 0.
- io_SerFlags. Set this to SERF_EOFMODE if you want CMD_READ to continue reading characters until it reaches one of the eight possible characters in the IOTArray structure.
- io_Length. Set this to the number of characters to be received from the Amiga serial port, or set it to −1 to tell the task to receive characters until an EOF character is read. Always specify io_Length as larger than the number of characters expected under the most extreme circumstances. If io_Length is too small, the task-defined read buffer will overflow and any RAM contiguous with the end of that buffer will be overwritten; the system may crash when it tries to access that RAM and finds information it cannot use or understand.
- io_Data. Set this to point to the task's read buffer, where characters coming in from the Amiga serial port will be placed. This is *not* the Serial device internal read buffer.

## Discussion

CMD_READ allows a task to place data into task-defined buffers coming from external hardware connected to the Amiga's serial port. Data is transferred from the external hardware to the serial-port data register, then to the Serial device internal read buffer, and eventually to the task-defined buffer. The Serial device internal read buffer acts as a temporary holding location for the data eventually destined for the task-defined buffer.

Data will continue to be read until the system detects a 0 or an IOTArray structure read termination character; a maximum of eight characters can be defined. This allows a task to tailor its read operations to each piece of external hardware. For example, if a device is known to put Ctrl-Z characters into its output stream to set off blocks of data, the task can define a character termination array consisting of eight Ctrl-Z characters (seven Ctrl-Z characters to dummy fill the array), allowing the task to stop reading after each block of data comes in from the external device.

## CMD_RESET

### Purpose of Command

CMD_RESET resets unit 0 to the boot-up time state as if it had just been initialized. All Serial device parameters and flag parameter bits are set to their default values. All active and queued CMD_READ, CMD_WRITE, and other I/O requests for Serial device unit 0 are aborted, and unit 0 is restarted if it was previously stopped by CMD_STOP. The current Serial device internal read buffer is relinquished, and a new read buffer with the default size (512 bytes) is allocated and initialized.

CMD_RESET is always executed as an immediate-mode command. All aborted I/O requests are replied to the task reply-port queue with the io_Error IOERR_ABORTED bit set. The results of command execution are found in the io_Error parameter. 0 indicates that the command was successful. SerErr_InvParam indicates that a task specified an invalid parameter in the IOExtSer structure used to define the CMD_RESET command. SerErr_NotOpen indicates that the Serial device has not yet been opened in the task, which should execute OpenDevice and CMD_RESET again.

### Preparation of the IOExtSer 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 Unit structures that manage Serial device unit 0. These can always be copied from the IOExtSer structure initialized by the OpenDevice function call. Also initialize io_Command to CMD_RESET, and set io_Flags to 0.

### Discussion

CMD_RESET is a destructive command. It calls CMD_FLUSH indirectly, thereby flushing all of the queued CMD_READ, CMD_WRITE, and other I/O requests in the unit 0 request queue. It also stops all ongoing CMD_READ and CMD_WRITE I/O requests dead in their tracks; the dispatching task then loses the data that it originally requested from the Serial device. CMD_RESET is always executed in immediate mode.

In addition, CMD_RESET indirectly calls CMD_START to start unit 0 if it was previously stopped with CMD_STOP. When a task once again starts to send Serial device I/O requests to unit 0, there will be no need to restart it. CMD_RESET also resets the CMD_RESET IOExtSer structure parameters and all flag bits to their default values. CMD_RESET does not affect any I/O requests currently queued in the task reply-port queue.

## CMD_START

### Purpose of Command

CMD_START restarts writes or reads if unit 0 was previously stopped by CMD_STOP. This includes any CMD_READ or CMD_WRITE command that was stopped in the middle of its activity, or the first CMD_READ or CMD_WRITE request at the top of the unit 0 device request queue when CMD_STOP was dispatched.

CMD_START performs these actions by reactivating the serial-port handshaking sequence. An XON character is sent to the other side, and a logical XON character is sent to the task's side of the transaction. For a CMD_WRITE command, the other side is the dispatching task; for a CMD_READ command, it is usually an external hardware device connected to the Amiga serial port.

CMD_START always operates as an immediate-mode command. The results of command execution are found in the io_Error parameter. 0 indicates that the command was successful. SerErr_InvParam indicates that a task specified an invalid parameter in the IOExtSer structure used to define the CMD_START command. SerErr_NotOpen indicates that the Serial device has not yet been opened in the task, which should execute OpenDevice and CMD_RESET again.

### Preparation of the IOExtSer 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 Unit structures that manage Serial device unit 0. These can always be copied from the IOExtSer structure initialized by the OpenDevice function call. Also initialize io_Command to CMD_START, and set the io_Flags parameter to 0.

### Discussion

CMD_START starts the reading and writing of data into or out of the serial-port data register. It is similar to the Ctrl-Q command used to restart screen output on most computers—it restarts execution of CMD_READ or CMD_WRITE commands previously stopped by CMD_STOP. CMD_START will also restart processing of queued I/O requests, just as Ctrl-Q will start to display additional files on the screen if the user has typed file-display commands.

The CMD_START command sends an XON character to the hardware device connected to the Amiga serial port. It also sends a logical XON character to the task that originally dispatched the CMD_READ or CMD_WRITE command. Both CMD_STOP and CMD_START will work only if the XON-XOFF protocol is enabled. This handshaking protocol is the default and can only be overridden by setting the IOExtSer structure io_SerFlags parameter SERF_XDISABLED bit. Note that the XON-XOFF protocol is the only handshaking protocol currently supported by the Serial device—it does not support the INQ-ACQ handshaking protocol.

Do not confuse the use of CMD_START with the use of the character termination array. CMD_START is designed to restart a CMD_READ or CMD_WRITE command previously stopped by CMD_STOP. The CMD_READ or CMD_WRITE command could have been dispatched originally with a character termination array designed into the data transfer definition.

## CMD_STOP

### Purpose of Command

CMD_STOP immediately stops a currently executing CMD_WRITE or CMD_READ command on unit 0. It also prevents the Serial device internal routines from starting execution of queued CMD_WRITE I/O requests. CMD_STOP is always an immediate-mode command.

CMD_STOP does its job by discontinuing the handshaking sequence for the serial port. The handshaking sequence sends an XOFF character to the other side and a logical XOFF character to the task's side of the transaction. For CMD_READ, the other side is the dispatching task; for CMD_WRITE, it is usually an external hardware device connected to the serial port. Once unit 0 is stopped by CMD_STOP, the system automatically queues CMD_READ, CMD_WRITE, and other I/O requests dispatched to unit 0 until CMD_START restarts unit 0 or CMD_RESET resets it.

The results of command execution are found in the io_Error parameter. 0 indicates that the command was successful. SerErr_InvParam indicates that a task specified an invalid parameter in the IOExtPar structure used to define the CMD_STOP command. SerErr_NotOpen indicates that the Serial device has not yet been opened in the task; you should execute OpenDevice and CMD_STOP again.

### Preparation of the IOExtSer 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 Unit structures that manage Serial device unit 0. These can always be copied from the IOExtSer structure initialized by the OpenDevice function call. Also initialize io_Command to CMD_STOP, and set io_Flags to 0.

### Discussion

The CMD_STOP command immediately stops execution of a CMD_READ or CMD_WRITE command. It is similar to the Ctrl-S command used for screen output on most computers—it stops a currently executing CMD_WRITE command at the earliest possible opportunity.

The CMD_STOP command sends an XOFF character to the hardware device connected to the serial port. It also sends a logical XOFF character to the current task that originally dispatched the CMD_READ or CMD_WRITE command. Both the CMD_STOP and CMD_START commands will only work if the XON-XOFF protocol is enabled. This handshaking mode is the default and can only be overridden by setting the IOExtSer structure io_SerFlags parameter SERF_XDISABLED bit. XON-XOFF protocol is the only handshaking protocol currently supported for the Serial device—it does not support the INQ-ACQ handshaking protocol.

Do not confuse the use of CMD_STOP with the use of the character termination array. CMD_STOP is designed to allow a task to stop an ongoing CMD_READ or CMD_WRITE command at any time. These commands could have been dispatched originally with a character termination array designed into the data transfer definition; EOF characters would stop the data transfer at any predefined character location.

## CMD_WRITE

### Purpose of Command

CMD_WRITE causes a stream of characters to be written from a task-defined buffer, one at a time, into the Serial device data register and out to an external hardware device connected to the serial port. The number of characters is specified in the IOExtSer structure io_Length parameter; if $-1$ is specified, the Serial device will write characters until an EOF character is written. The system default EOF character is 0.

If the io_SerFlags parameter SERF_EOFMODE bit is set, CMD_WRITE will continue to write characters until the first EOF character defined by the IOTArray structure is encountered in the task-defined buffer data. A CMD_WRITE command can be terminated early if a write error occurs or if an EOF condition is encountered; in this case, the number of characters written is placed into the IOExtSer structure io_Actual parameter.

CMD_WRITE can be treated as a synchronous or an asynchronous I/O request. If a CMD_WRITE command is specified as QuickIO and is unsuccessful, the I/O request will be replied to the calling task reply-port queue.

The results of command execution are found in the io_Actual and io_Error parameters. io_Actual indicates the number of characters actually written. The Serial device internal routines set this value if an error or an EOF character was encountered during the writing process. The count does not include the termination character, even though it is written out to the external hardware device.

A 0 value in the io_Error parameter indicates that the command was successful. Other io_Error values have the following meanings:

- SerErr_DevBusy. The Serial device was busy and could not execute CMD_WRITE as requested.
- SerErr_InvParam. A task specified an invalid parameter in the IOExtSer structure used to define CMD_WRITE.
- SerErr_LineErr. There was a line error during the read operation.
- SerErr_NotOpen. The Serial device has not yet been opened in the task, which should execute OpenDevice and CMD_WRITE again.
- SerErr_InvBaud. The specified baud rate is invalid (usually out of range); change the task-specified baud rate and dispatch CMD_WRITE again.
- SerErr_ParityErr. A parity error occurred during data transfer; the task should dispatch CMD_WRITE again.
- SerErr_TimerErr. A timing error occurred during data transfer; the task should dispatch CMD_WRITE again.
- SerErr_NoDSR. No DSR (data set ready) signal was sent during data transfer; the task should determine why and dispatch CMD_WRITE again.
- SerErr_NoCTS. No CTS (clear to send) signal was sent during data transfer; the task should determine why and dispatch CMD_WRITE again.
- SerErr_DetectedBreak. The system detected a queued or immediate break signal during data transfer; the task should eliminate the break signal and dispatch CMD_WRITE again.

## Preparation of the IOExtSer 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 Unit structures that manage Serial device unit 0. These can always be copied from the IOExtSer structure initialized by the OpenDevice function call. Also set the following parameters:

- io_Command. Set this to CMD_WRITE.
- io_Flags. Set this to IOF_QUICK for QuickIO; otherwise, set it to 0.
- io_SerFlags. Set this to SERF_EOFMODE if you want CMD_WRITE to continue writing characters until it writes one of the eight possible IOTArray structure EOF characters.
- io_Length. Set this to the number of characters to send to the serial port. Set it to −1 to tell the task to receive characters until an EOF character is read out to the serial-port data register from the task-defined write buffer.

- io_Data. Set this to point to the task's write buffer, which contains the characters that will be sent to the serial-port data register.

## Discussion

CMD_WRITE allows a task to send data from a task-defined buffer out to the serial-port data register and eventually out to external hardware devices connected to the Amiga serial port. Data is transferred one character at a time. It will continue to be written until the system detects a 0 character in the data or until it detects a write termination character defined in the IOTArray structure. A maximum of eight characters can be defined. If the io_SerFlags SERF_EOFMODE bit is set, the system will continue writing characters until a termination character is detected in the output stream.

This arrangement allows a task to tailor its write operations to each piece of external hardware. For example, if a physical device connected to the serial port requires blocks of data separated by Ctrl-Z characters, the task can define a character termination array consisting of eight Ctrl-Z characters. A large task-defined buffer can then be set up with Ctrl-Z characters between blocks of data in that buffer. Then, when CMD_WRITE is executed, the task will stop writing after each block of data is written out to the external device. The task can execute a number of CMD_WRITE commands, and each of these can send a new block of data out to the external device. The Ctrl-Z characters will also be written, allowing the external device to properly format its data.

A Serial device input buffer is used for CMD_READ execution, but not for CMD_WRITE command execution. For CMD_WRITE, data passes directly from the task-defined buffer through the Serial device data register and out to the external hardware device.

## DEVICE-SPECIFIC COMMANDS

## *SDCMD_BREAK*

### Purpose of Command

The SDCMD_BREAK command sends a break signal to the serial port line, causing it to be held electrically low for an extended time. This is accomplished by setting the UARTBRK bit of the Serial device ADKCON register. A task specifies the time period by setting the IOExtSer structure io_BrkTime parameter using SDCMD_SETPARAMS. The default length of the break signal is 250,000 microseconds. After the stated time, the UARTBRK bit is reset and the break signal is discontinued.

If the IOExtSer structure io_SerFlags SERF_QUEUEDBRK bit is set, the SDCMD_BREAK request is placed at the bottom of the Serial device unit 0 request queue, and SDCMD_BREAK is executed when it works its way to the top of the queue. If SERF_QUEUEDBRK is not set, the break signal is started immediately; control returns to the dispatching task and the timer discontinues the break signal after the specified duration has expired. If the IOExtSer structure io_Flags parameter IOF_QUICK bit is cleared, the dispatching task will receive a reply when the break signal is completed.

The results of command execution are found in io_Error. A 0 value indicates that the command was successful. SerErr_InvParam indicates that a task specified an invalid IOExtPar structure parameter to define the SDCMD_BREAK command. SerErr_NotOpen indicates that the Serial device has not yet been opened in the task, which should execute OpenDevice and SDCMD_BREAK again.

## Preparation of the IOExtSer 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 Unit structures that manage Serial device unit 0. These can always be copied from the IOExtSer structure initialized by the OpenDevice function call. Also initialize io_Command to SDCMD_BREAK, and set io_Flags to 0.

## Discussion

SDCMD_BREAK allows a task to send a break signal out through the serial port line to an external hardware device. It can be an immediate-mode command. The actual duration of the break signal is defined by the IOExtSer structure io_BrkTime parameter, which can be specified using either the SDCMD_SETPARAMS command or a simple structure-parameter assignment statement. Once the io_BrkTime parameter is initialized, it controls the break signal duration for all SDCMD_BREAK commands dispatched from then on or until another value is specified.

If the io_SerFlags SERF_QUEUEDBRK bit is set, SDCMD_BREAK commands will be queued in the unit 0 request queue and will be executed in the order in which they were queued. If SERF_QUEUEDBRK is cleared before the SDCMD_BREAK command is dispatched, the command is executed immediately. In fact, it may interrupt a currently active command request. When it has finished execution, the interrupted request will continue where it left off, provided the I/O request has not been aborted.

Two bits in the IOExtSer structure io_Status parameter report the status of break signals. Bit 1 reports the break signal sent, and bit 2 reports the break signal received. A task can always look at these two bits to determine current conditions for break signals in the system. The dispatching task must properly coordinate its SDCMD_BREAK commands with other commands and functions in the Serial device system. You should be aware that SDCMD_BREAK can interact with AbortIO, CMD_FLUSH, CMD_STOP, and CMD_START.

## SDCMD_QUERY

### Purpose of Command

The SDCMD_QUERY command allows a task to determine the current status of the Serial device internal routines and all hardware involved in the Serial device system. A task can use SDCMD_QUERY to determine if the Serial device routines are currently reading or writing to a hardware device—in particular, to a serial modem. The IOExtSer structure io_Status parameter is kept up-to-date as events in the system change the status of hardware and software.

SDCMD_QUERY is an immediate-mode command. The results of command execution are found in the io_Error and io_Status parameters. The bits in the IOExtSer structure io_Status parameter return the values shown previously in Table 6.2. A 0 value in io_Error indicates that the command was successful. SerErr_InvParam indicates that a task specified an invalid parameter in the IOExtSer structure used to define the SDCMD_QUERY command. SerErr_NotOpen indicates that the Serial device has not yet been opened in the task, which should execute OpenDevice and SDCMD_QUERY again.

### Preparation of the IOExtSer 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 Unit structures that manage Serial device unit 0. These can always be copied from the IOExtSer structure initialized by the OpenDevice function call. Also initialize io_Command to SDCMD_QUERY, and set the io_Flags parameter to 0.

### Discussion

The SDCMD_QUERY command was included in the Serial device software system to allow a task to monitor the internal activity of the Serial device and the external hardware connected to the serial port. With this command, you can decide what to do next based on the current status and activity of the Serial device.

SDCMD_QUERY is usually used to monitor the behavior of a modem attached to the serial port and driven by CMD_READ and CMD_WRITE commands coming from a task. The task needs to know about the modem's external conditions—if the modem is currently working with the task, if it is busy sending or receiving data, or if it is experiencing a data transfer error. The IOExtSer structure io_Status parameter determines the current status of the modem-task data transfer. In addition, the task must know whether the Serial device is reading or writing to the external hardware device. This information is provided by the ten active bits of the io_Status parameter.

## SDCMD_SETPARAMS

### Purpose of Command

SDCMD_SETPARAMS allows a task to set the Serial device parameters. It will only be successful if there are no active or queued CMD_READ or CMD_WRITE I/O requests already dispatched. The only exception to this rule is a SDCMD_SETPARAMS command that attempts to change the handshaking protocol by enabling or disabling the XON-XOFF protocol using the io_SerFlags SERF_XDISABLED bit.

SDCMD_SETPARAMS is an immediate-mode command. It is usually used when a task changes IOExtSer structure parameters (io_ExtFlags, io_SerFlags, and io_TermArray) before dispatching a CMD_READ or CMD_WRITE command. These parameters can be initialized or reinitialized with the SDCMD_SETPARAMS command.

The results of command execution are found in io_Error. 0 indicates that the command was successful. SerErr_InvParam indicates that a task specified an invalid parameter in the IOExtSer structure used to define SDCMD_SETPARAMS. SerErr_NotOpen indicates that the Serial device has not yet been opened in the task; it should execute OpenDevice and dispatch SDCMD_SETPARAMS again.

### Preparation of the IOExtSer 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 Unit structures that manage Serial device unit 0. These can always be copied from the IOExtSer structure initialized by the OpenDevice function call. Set io_Command to SDCMD_SETPARAMS.

Also initialize the following parameters:

- io_Flags. Set this to 0.
- io_CtlChar. Initialize this to a longword (four bytes) containing byte values for the XON and XOFF parameters. (INQ and ACK protocol parameters are not used as of Release 1.2.)
- io_RBufLen. Initialize this to the size of the Serial device internal read buffer. (This is *not* the size of the task-defined buffer—io_Data points to that.) The read buffer must be at least 512 bytes long; otherwise, a buffer overflow error could occur and the system might crash. Any change in the io_RBufLen parameter causes the old Serial device internal read buffer to be deallocated and a new, correctly sized buffer to be allocated. Therefore, the contents of the old buffer are lost.
- io_ExtFlags. Set this to 0; it is not used at present.
- io_Baud. Set this to the baud rate used for CMD_READ and CMD_WRITE commands. The allowed range is from 110 to 29,200 baud. Asynchronous I/O

above 32K baud may not be successful, especially on a busy system; the Serial device may miss bits when task-switching occurs.

- io_BrkTime. Set this to the duration of the break signal in microseconds; the default value is 250,000 microseconds.
- io_TermArray. Set this to point to a descending ASCII order 8-byte array of termination characters for the task. This parameter is initialized by a new OpenDevice function call to reflect the current state and configuration of the Serial device routines only if the SERF_EOFMODE io_SerFlags flag parameter bit is set in the IOExtSer structure used for the OpenDevice function call.
- io_ReadLen. Set this to the number of bits (1–8) in each word read by the CMD_READ command. This does not include the parity bit.
- io_WriteLen. Set this to the number of bits (1–8) in each word written by the CMD_WRITE command. This does not include the parity bit.
- io_StopBits. Set this to the number of bits (0, 1, or 2) in each word read or written by CMD_READ or CMD_WRITE. The normal value is one stop bit. Two stop bits can be used for a CMD_READ command if the io_ReadLen parameter is 7.
- io_SerFlags. Set this to SERF_SHARED if you want to open the Serial device in shared access mode in a subsequent OpenDevice function call; set it to 0 for exclusive access mode, the default. Set io_SerFlags to SERF_RAD_BOOGIE to access the Serial device in high-speed mode. Set io_SerFlags to SERF_EOFMODE if you want to use a set of task-defined EOF characters to control end-of-file conditions for CMD_READ and CMD_WRITE commands.

  The io_SerFlags parameter bits are always initialized by a new OpenDevice function call to reflect the current state and configuration of the Serial device internal routines as determined by the default values or previous parameter settings. The default values for the io_SerFlags parameter are XON-XOFF enabled, parity checking invoked, 3-WIRE protocol active, and the IOTArray character termination array inactive.

## Discussion

SDCMD_SETPARAMS allows a task to directly change the Serial device parameters used by subsequent CMD_READ and CMD_WRITE command I/O requests. It is usually used to set Serial device parameters before an OpenDevice function call, or after an OpenDevice function call but before the next CMD_READ or CMD_WRITE command is dispatched. SDCMD_SETPARAMS and the OpenDevice function interact throughout a task's execution.

SDCMD_SETPARAMS is particularly important in defining characters for terminating a CMD_READ or CMD_WRITE command I/O request. It allows a task to change the current end-of-file character definition for the next CMD_READ or CMD_WRITE command. If a task is dealing with a number of hardware devices with different data characteristics (for example, different character-block termination characters), it can change the current character termination array so that the next CMD_READ or CMD_WRITE can relate to the next hardware device with which it wants to communicate.

You can also define a separate task for each Serial device attached to the Amiga's serial port. Separate character termination arrays would then be specified for each task in order for it to work with the specific device characteristics and block definitions.

XON-XOFF is the default protocol for data transfers. SERF_XDISABLED is the only flag parameter bit that can be changed while the Serial device internal routines are actively processing a CMD_READ or CMD_WRITE command. If a task changes the SERF_XDISABLED bit in this way, the CMD_READ or CMD_WRITE IOExtSer structure will be replied with the io_Error parameter set to SerErr_DevBusy.

Keep in mind that the IOExtSer structure io_SerFlags SERF_EOFMODE and SERF_QUEUEDBRK flag parameter bits can be set or reset directly without a SDCMD_SETPARAMS command. The io_SerFlags SERF_SHARED and SERF_7WIRE bits can also be reinitialized directly using simple structure-parameter assignment statements before an OpenDevice function call. All other Serial device parameters can only be changed by sending a SDCMD_SETPARAMS command.

If your task is trying to run an MIDI (musical instrument digital interface), it should set the io_SerFlags SERF_RAD_BOOGIE bit, which automatically sets the SERF_XDISABLED bit to eliminate unneeded data transfer overhead, thereby allowing the fastest possible data transfer. SERF_RAD_BOOGIE also checks for parity, XOFF parity character-handling, and specified character lengths other than eight bits. It also tests for a break signal.

Writing MIDI formatted data at MIDI rates is easily accomplished. However, using the Serial device routines for MIDI alone may not be reliable because of MIDI time-stamping requirements and data overruns in a busy multitasking system or display environment, which can steal CPU time away from the Serial device routines and cause data transfer characters to be missed.

Selecting mark or space parity will cause the SERF_PARTY_ON (parity-checking enabled) bit to be set and the SERF_PARTY_ODD (odd parity) bit to be ignored.

# 7 The Input Device

## Introduction

The Input device automatically merges input events from many sources—from the Keyboard, Gameport, Timer, and TrackDisk devices, and from task-defined input events. It does not process input events; it only merges them into a properly linked input event stream, which is then passed to a set of input-handler functions defined by the programmer. The input-handler functions then process the input events.

The Input device is ROM-resident and is loaded when the WCS ROM code and data are loaded from the Kickstart disk. It is opened automatically if a task executes a call to open the Console device. When the Input device is opened, the system automatically starts a task (referred to as the input task), which communicates directly with the appropriate devices to obtain raw key events, mouse-button and mouse-movement events, timer events, and disk insertion and removal events.

The input task also communicates directly with a programmer-defined task to merge any task-defined input events into the total input event stream. When the input task is operating, it attempts to intercept and coordinate Keyboard, TrackDisk, Timer, and Gameport device input events.

In general, there are three ways to deal with the separate categories of input events. The first method is to use commands and functions to process distinct input events from each of the separate sources. The second method is to use a set of programmer-defined input-handler functions to preprocess the input event stream, as you will see later in this chapter.

The third method is to pass the input event stream to the Console device—or to Intuition, which can then use an IDCMP to deal with the stream. If you use this method, you should be aware of what happens to input events if your task does not respond to them. If there is no currently active window or console unit, the input events (keystrokes or left mouse-button clicks) will be ignored by Intuition. If your task has opened an Intuition window and made it active, and if you choose to let the input events queue without responding to them, the following will happen:

- Another input event will occur. If the system has no empty InputEvent structure that it can use to represent the new input event, it will automatically allocate RAM for a new InputEvent structure, which will be queued in the message port for the task.

- When the task finally responds to the input event, the memory allocated for the InputEvent structure will not be returned to the system free-memory pool until the active window is closed. Therefore, a task that does not respond to its incoming input events for a long period of time can remove a great deal of memory from the system. (Of course, it is wise to program your tasks to respond to input events as quickly as possible to maximize free memory at all times.)

## Operation of the Input Device

Figure 7.1 illustrates the operation of the Input device. As the figure illustrates, the Input device can merge input from five sources:

- The Keyboard device, which is always open and active, sends the Input device a continuous stream of keyboard events. Each time a key is pressed, the Keyboard device generates a new input event for the Input device to process.
- The TrackDisk device, which is always open and active, sends the Input device an input event every time the user inserts or removes a disk from a disk drive.
- The Timer device, which is always open and active, sends the Input device a continuous stream of input events; these are system times represented by individual TimeVal structures.
- The Gameport device, if open and active, sends the Input device a continuous stream of gameport events. Each time the mouse (or another controller) is moved, a new input event is generated.
- Each task in the system can create a set of InputEvent structures for the Input device to process; in this way a programmer can, for example, simulate the input events generated by hardware devices.

The Input device merges input events from these five sources into one input event stream. Each input event is represented by an InputEvent structure containing a TimeVal structure characterizing when it was created (in other words, when the input event occurred). Each new input event is added to the bottom of the input event list. The InputEvent structure ie_NextEvent parameter links input events together to form the total stream. An input event is entered into the stream according to the time at which it was generated.

The Input device internal routines link and organize all input events automatically (except for some task-defined input events, as described in the next section). The merged stream is then fed to a series of input-handler functions, which filter the stream into various categories that can be processed by other tasks and routines in the system.

## Using Input-Handler Functions

Input-handler functions allow a task to define a set of functions that the Input device software system automatically uses to preprocess all input events coming into the system. A task can design its input-processing operations to deal with input events in any way that accomplishes its goals. For example, an input-handler function could be designed to filter all keyboard input events and change them into other keyboard input events (to change A's to B's or B's to A's, for example); note, however, that this specific type of keyboard preprocessing is usually handled directly by the Keyboard device commands and functions (see Chapter 9). A programmer could also design an input-handler function to preprocess

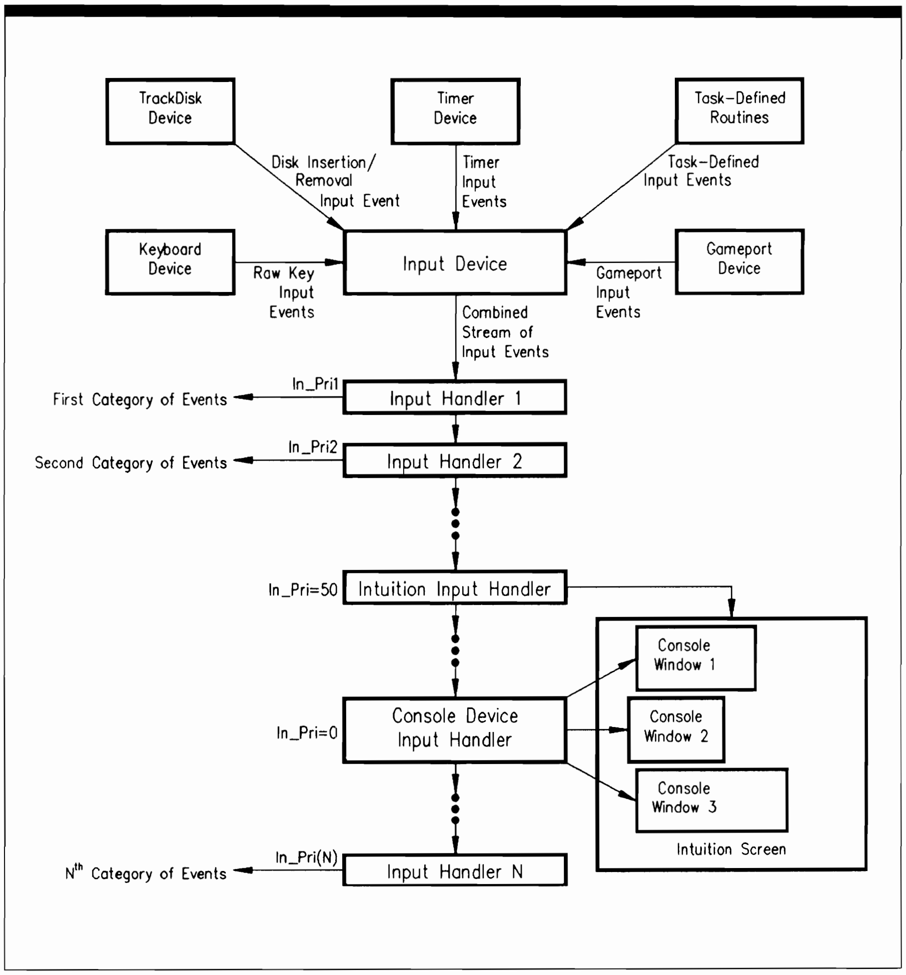


**Figure 7.1:** *Operation of the Input Device*

input events coming from a gameport to which equipment other than the standard Amiga mouse (for example, a digitizer) was connected. Each input-handler function should be designed to preprocess a specific subset of input events.

Input-handler functions are added to the system with the IND_ADDHANDLER command. An input-handler function is designed within the Exec Interrupt structure framework, which defines the executable code and the data needed by that code. Each function is added to the input-handler function list; it will preprocess a prescribed subset of input events until it is removed by the IND_REMHANDLER command. Because IND_ADDHANDLER can install an input-handler function at any position in the function list, that function can ignore certain types of input events and also act upon and modify other types of input events in the stream. It can even create new input events for Intuition or other programs.

An input-handler function is assigned a priority (in the Interrupt structure Node substructure ln_Pri parameter) when IND_ADDHANDLER adds it to the input-handler function list. This priority determines where the function is positioned in the list. If the priority parameter is initialized to −128, the function will be placed at the bottom of the list and will be the last to preprocess input events. If the priority parameter is initialized to 127, the input-handler function will be placed at the top of the list and will be the first to preprocess input events.

Note that the input-handler function for the Intuition software system is placed at priority 50. Therefore, if you want your task to preprocess keyboard or mouse events (or other events normally processed by Intuition), you should create an input-handler function with a priority greater than 50.

At any given time, the current highest-priority input handler has access to the entire list of input events; it will generally preprocess and therefore eliminate some of these. Input events that are not preprocessed by the highest-priority input handler are passed on to the input-handler function with the next-highest priority. This process continues until some or all input events have been preprocessed and eliminated by the input-handler functions.

Preprocessing is continuous; input events come into the system from the input sources, and at the same time, the input event stream is being fed to the linked list of input-handler functions. In addition, tasks can be adding and removing input-handler functions from the function list in order to preprocess input events properly and to satisfy their needs. This dynamic and continuously changing process is mostly under the control of the system and Input device internal routines.

You should always observe the following rules when you design an input-handler function:

- If you want a specific type of input event that has no link to other input events in the stream (the InputEvent structure ie_NextEvent parameter is null), your input-handler function can end the function list by returning a null value. (Note that this returned null value is the value returned by the input-handler function, not the null ie_NextEvent parameter.) No other input event will be sent to lower-priority input-handler functions in the function list.

- If the input event stream consists of multiple input events linked together, your input-handler function can be designed to delink an input event from the input event stream. This is done by passing a shorter list of input events to all lower-priority input-handler functions in the input-handler function list. In this case, the function should be designed to return the address of the InputEvent structure defining the first input event in the shorter input event stream. All lower-priority input-handler functions will then work with the shortened list.

- If you want to add new input events to the stream that your function passes to lower-priority functions, you can add new InputEvent structures to RAM to define them. When your function gets control on the next round of input event handling, it should assume nothing about the current contents of those InputEvent structures;

higher-priority input-handler functions may have modified them. Your input-handler function should also keep track of the starting address and the amount of RAM used for the InputEvent structures. It should free the InputEvent structure memory blocks so that memory can be returned to the system when it is no longer needed.

## Input Device Commands

You program the Input device with eight device-specific commands; only four standard device commands are supported. The command discussions in this chapter indicate which commands support QuickIO, queued I/O, and immediate-mode operation. All Input device commands affect either the IOStdReq or the TimeRequest structure io_Error parameter.

### Sending Commands to the Input Device

Figure 7.2 depicts the general scheme used to dispatch commands to the Input device routines. The lines with arrows represent the parameters you should initialize and also those returned by the device internal routines.

The Input device programming process consists of three phases:

1. Structure preparation. You have complete control over this phase—here, you initialize parameters in the IOStdReq, TimeRequest, and Interrupt structures in preparation for dispatching a command to the Input device internal routines. These parameters include the usual set of parameters required by most devices. In addition, you must initialize the Interrupt structure is_Data and is_Code pointer parameters referenced by the IND_ADDHANDLER and IND_REMHANDLER commands and the tv_Secs and tv_Micros parameters for the IND_SETPERIOD and IND_SETTHRESH commands. The choice of parameters to initialize depends on the specific command you plan to send to the Input device. All of these parameters taken together provide an information path to the data needed by the Input device routines to process the command.

2. Input device processing. The only part you play in this phase is to send a command to the device using the BeginIO, DoIO, or SendIO function. Once the function begins executing, control passes to a mixture of device and system internal routines.

3. Command output parameter processing. The system and Input device routines have complete control over the values found in these parameters. The results of Input device command processing have been returned to the task that originally issued the command. If the I/O request was not QuickIO or immediate-mode, it was processed when it moved to the top of the Input device request queue. The Input device then replied and the I/O request was returned to the task reply-port queue to await the task's processing. If the request was a successful QuickIO, it was not queued in the task reply-port queue but came back directly to the requesting task with the io_Flags IOF_QUICK bit still set. These five parameters still direct you to appropriate data for your task.

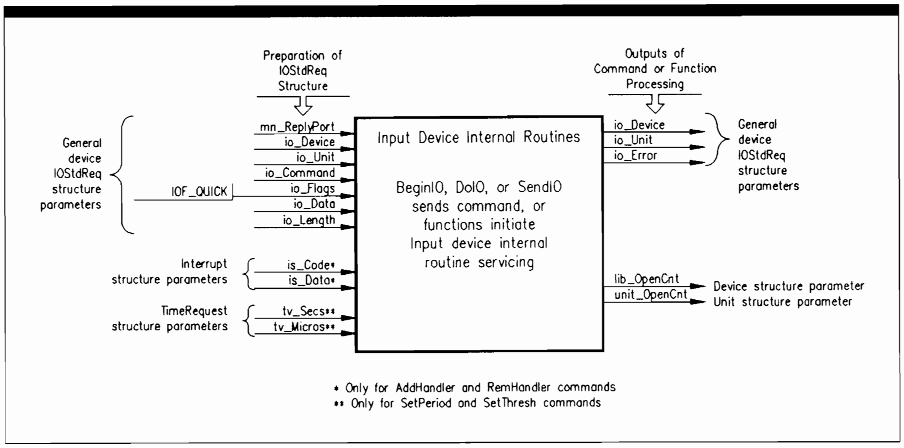

Preparation of IOStdReq Structure
Outputs of Command or Function Processing
General device IOStdReq structure parameters
mn_ReplyPort
io_Device
io_Unit
io_Command
IOF_QUICK
io_Flags
io_Data
io_Length
Interrupt structure parameters
is_Code*
is_Data*
TimeRequest structure parameters
tv_Secs**
tv_Micros**
Input Device Internal Routines
BeginIO, DoIO, or SendIO sends command, or functions initiate Input device internal routine servicing
io_Device
io_Unit
io_Error
General device IOStdReq structure parameters
lib_OpenCnt
unit_OpenCnt
Device structure parameter
Unit structure parameter
* Only for AddHandler and RemHandler commands
** Only for SetPeriod and SetThresh commands

**Figure 7.2:** *Input Device Command and Function Processing*

The device internal routines do not directly return any output structure parameters for any of the Input device commands. You can contrast this to other device commands, most of which do return values.

Figure 7.2 also depicts the parameters that play a part in Input device function setup and processing. The OpenDevice and CloseDevice functions both affect the unit_OpenCnt parameter and lib_OpenCnt parameter in the Unit and Device structures; OpenDevice also affects the io_Error parameter and returns an io_Error value.

## Structures for the Input Device

Figure 7.3 illustrates the structures required to operate the Input device. The Input device deals directly only with the InputEvent structure. However, the IOStdReq, TimeRequest, and Interrupt structures are used to define information necessary for some Input device uses, so they are included in the figure and in this discussion.

The InputEvent structure defines the characteristics of the input events. It contains two substructures: ie_xy, which is actually defined in the InputEvent structure, and a TimeVal structure named ie_TimeStamp, which is used to record the time at which the input event occurred. To save memory, the InputEvent structure contains a union that holds position or address information required to characterize the input event. (The Input device system may queue a great number of input events before it can process them; any savings in memory can prevent a memory overrun from occurring.) The InputEvent structure also contains the ie_NextEvent pointer, which points to the next input event in a linked list. For example, if mouse movements generate a large number of input events, they will be maintained in a linked list using this pointer.

The Interrupt structure is usually used to define a software interrupt in the Amiga system. For the Input device, it is used by the IND_ADDHANDLER and IND_REMHANDLER commands to describe a new input-handler function. The

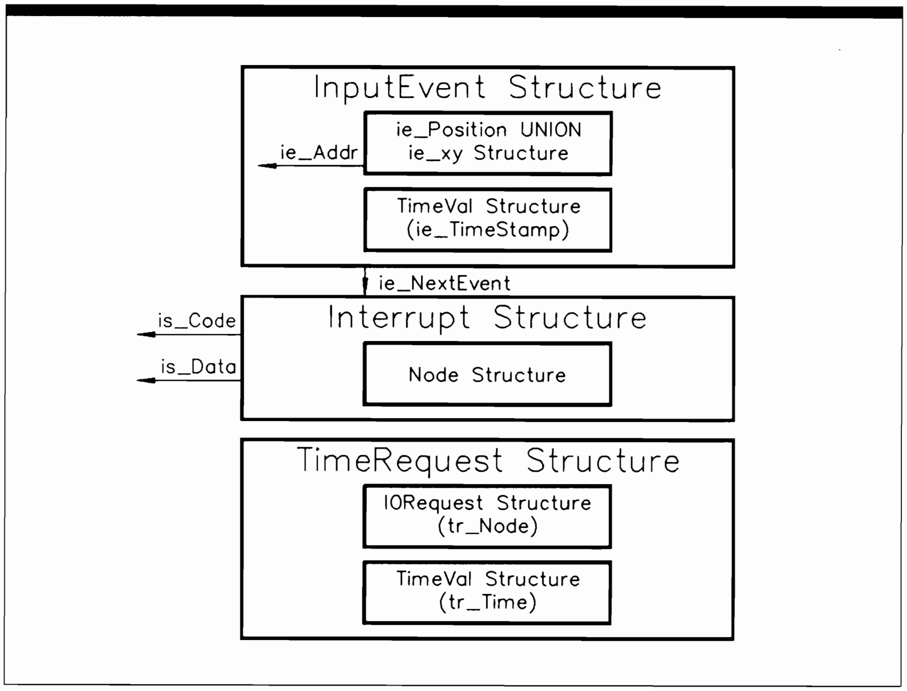


**Figure 7.3:** *Input Device Structures*

is_Code and is_Data parameters define the code and data used by an input-handler function when it preprocesses input events before passing them on to other input-handler functions. For standard Input device commands, the IOStdReq structure is used to define I/O requests.

The TimeRequest structure consists of an IORequest substructure and a TimeVal substructure. It is used as the I/O request structure for the IND_SETPERIOD and IND_SETTHRESH commands to set the characteristics of the Amiga keyboard; see Chapter 13 for a full discussion.

## The InputEvent Structure

The InputEvent structure is defined as follows:

```
struct InputEvent {
   struct InputEvent *ie_NextEvent;
   UBYTE ie_Class;
   UBYTE ie_SubClass;
   UWORD ie_Code;
   UWORD ie_Qualifier;
   union {
   struct {
```

```
            WORD ie_x;
            WORD ie_y;
        } ie_xy;
        APTR ie_addr;
        } ie_position;
        struct TimeVal ie_TimeStamp;
    };
```

The InputEvent parameters have the following meanings:

- ie_NextEvent points to an InputEvent structure representing the next input event in the linked list. This is the next event chronologically, no matter where it originated.
- ie_Class is the class of the input event.
- ie_SubClass is the subclass of the input event (optional).
- ie_Code is the input event code.
- ie_Qualifier represents all input event qualifiers currently in effect; they are discussed below.
- ie_x is the X position of the mouse pointer in the window when the mouse input event occurs.
- ie_y is the Y position of the mouse pointer in the window when the mouse input event occurs.
- ie_xy is the name of the mouse position substructure in the ie_position union.
- ie_addr is the address of the input event.
- ie_position is the name of the InputEvent structure union.
- ie_TimeStamp is the name of a TimeVal substructure inside the InputEvent structure; it stores the timestamp (time of occurrence) of the input event.

The input event qualifiers (found in the ie_Qualifier parameter) describe the following input events:

- IEQUALIFIER_LSHIFT. The left Shift key was pressed.
- IEQUALIFIER_RSHIFT. The right Shift key was pressed.
- IEQUALIFIER_CAPSLOCK. The Caps Lock key was pressed.
- IEQUALIFIER_CONTROL. The Ctrl key was pressed.
- IEQUALIFIER_LALT. The left Alt key was pressed.
- IEQUALIFIER_RALT. The right Alt key was pressed.
- IEQUALIFIER_LCOMMAND. The left Amiga key was pressed.

- IEQUALIFIER_RCOMMAND. The right Amiga key was pressed.
- IEQUALIFIER_NUMERICPAD. A numeric pad key was pressed.
- IEQUALIFIER_REPEAT. The previous keyboard input event was repeated.
- IEQUALIFIER_INTERRUPT. A system interrupt occurred.
- IEQUALIFIER_MULTIBROADCAST. The event will be sent to all open Intuition windows, not just the currently active window.
- IEQUALIFIER_LBUTTON. The left mouse button was pressed.
- IEQUALIFIER_RBUTTON. The right mouse button was pressed.
- IEQUALIFIER_MBUTTON. The middle mouse button was pressed. (This is not available on the standard Amiga mouse.)
- IECLASS_RELATIVEMOUSE. The mouse coordinates are relative positions, not absolute positions.

For more information on input event classes, codes, and qualifiers, see the Inputevent.h INCLUDE file.

## USE OF FUNCTIONS

## *CloseDevice*

### Syntax of Function Call

```
CloseDevice (iOStdReq)
                A1
```

### Purpose of Function

This function closes access to Input Device unit 0. If this is the last CloseDevice function call for the Input device and the Console device is also closed, the Timer, Keyboard, and Gameport devices will also be closed by it. CloseDevice also decrements the Device structure lib_OpenCnt parameter and the Unit structure unit_OpenCnt parameter, reducing each by 1. If a deferred expunge for the Input device is pending, its routines are expunged from ROM as soon as these parameters are reduced to 0 and CloseDevice returns.

When CloseDevice returns, the current task cannot use the Input device until it executes another OpenDevice function call or until the Console device is opened in the same task. Current Input device parameters are saved for the next call to OpenDevice.

### Inputs to Function

| | |
|---|---|
| **iOStdReq** | A pointer to an IOStdReq structure |

### Discussion

CloseDevice provides a way to terminate access to a set of device routines. Because unit 0 is the only valid unit of the Input device, the CloseDevice function always closes access to the device as a whole in the current task.

The Input device is a shared access mode device; a number of tasks can access it at one time. However, to conserve system memory, you may decide to have each task that opens the Input device call CloseDevice before another task calls OpenDevice.

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

CloseDevice also closes the Timer, Keyboard, and Gameport devices in the current task automatically. However, since they are all shared access mode devices, they can remain open in other tasks that have either opened them explicitly with OpenDevice or opened them indirectly through the Input or Console device.

The system input task opens the Input device as part of the machine startup sequence. If the system closes the Input device, input events cannot be processed; the machine will not respond to any keyboard or gameport events.

## *OpenDevice*

### Syntax of Function Call

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

### Purpose of Function

This function opens access to the internal routines of Input device unit 0. It also opens the Timer, Gameport, and Keyboard devices if they have not already been opened in the task. The Input device is always opened in shared access mode.

OpenDevice increments the Device structure lib_OpenCnt parameter and the Unit structure unit_OpenCnt parameter, thereby preventing a deferred expunge of the Input device. It requires a properly initialized mn_ReplyPort parameter with a task signal bit allocated to that message reply port if the calling task wants to be signaled. The results of function execution are as follows:

- io_Device. This points to a device structure that manages Input device unit 0 once it is opened.
- io_Unit. This points to a Unit structure used to define and manage a MsgPort structure for Input device unit 0. The MsgPort structure represents the device request queue.
- io_Error. A 0 value indicates that the requested open succeeded. IOERR_OPEN-FAIL indicates that the Input device could not be opened; this is usually caused by lack of memory.

## Inputs to Function

| | |
|---|---|
| **"input.device"** | A pointer to a null-terminated string representing the name of the Input device |
| **0** | The Input device unit number |
| **iOStdReq** | A pointer to an IOStdReq structure |
| **0** | Indicates that the flags argument is not used |

## 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 from a previous OpenDevice call. Set io_Command to 0, or set it to IND_WRITEEVENT if the task should open the Input device and dispatch an IND_WRITEEVENT 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 Input device, no typecasting is necessary.

## Discussion

OpenDevice can be called with appropriate parameters to open the Input device and to initialize parameters to define an IND_WRITEEVENT command. Once a task owns the Input device, it can dispatch a series of IND_WRITEEVENT commands (with BeginIO, DoIO, or SendIO) to define its own input events to the Input device internal routines.

STANDARD DEVICE COMMANDS

## CMD_FLUSH

### Purpose of Command

The CMD_FLUSH command aborts all queued command requests currently in the Input device request queue; active Input device command requests are not affected. All aborted I/O requests are replied to the task reply-port queue with the io_Error IOERR-_ABORTED bit set.

CMD_FLUSH is always executed as an immediate-mode command. The results of command execution are found in io_Error, where 0 indicates that the command was successful. IOERR_NOCMD indicates that the task specified the io_Command parameter incorrectly; IOERR_ABORTED indicates that the specified command was aborted with Abort_IO or CMD_FLUSH.

### 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 Unit structures that manage Input device unit 0. These can always be copied from the IOStdReq structure initialized by the OpenDevice function call. Also initialize io_Command to CMD_FLUSH and set io_Flags to 0.

### Discussion

The CMD_FLUSH command flushes all currently pending command requests from the unit 0 device request queue. Because it is a destructive command, you should use it only if you want to restore the system to some known state with an empty Input device request queue. CMD_FLUSH does not directly affect the state of task reply-port queue, in which previously replied command requests may be queued.

## CMD_RESET

### Purpose of Command

CMD_RESET resets unit 0, returning all Input device internal parameters and flag parameter bits to their default values. CMD_RESET aborts all active and queued I/O requests for unit 0 and calls CMD_START to start the unit if it was stopped previously with CMD_STOP. All aborted I/O requests are replied to the task reply-port queue with the io_Error IOERR_ABORTED bit set.

CMD_RESET is always executed as an immediate-mode command. The results of command execution are found in io_Error, where 0 indicates that the command was successful. IOERR_NOCMD indicates that the task specified the io_Command parameter 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 Device and Unit structures that manage Input device unit 0. These can always be copied from the IOStdReq structure initialized by the OpenDevice function call. Also initialize io_Command to CMD_RESET and set io_Flags to 0.

### Discussion

CMD_RESET is a destructive command. It calls CMD_FLUSH indirectly, thereby flushing all of the queued command I/O requests in the unit 0 device request queue. It also stops all current command I/O requests.

## *CMD_START*

### Purpose of Command

If unit 0 was previously stopped by the CMD_STOP command, CMD_START restarts it, including any command that was stopped in the middle of its activity and the first command at the top of the unit 0 device request queue when CMD_STOP was dispatched.

CMD_START is always executed as an immediate-mode command. The results of command execution are found in io_Error, where 0 indicates that the command was successful. IOERR_NOCMD indicates that the task specified the io_Command parameter incorrectly. IOERR_ABORTED indicates that the specified command was aborted with AbortIO or CMD_FLUSH.

### 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 Unit structures that manage Input device unit 0. These can always be copied from the IOStdReq structure initialized by the OpenDevice function call. Also initialize io_Command to CMD-_START and set io_Flags to 0.

### Discussion

CMD_START is similar to the Ctrl-Q command used to restart screen output on most computers—it restarts execution of commands previously stopped by the CMD_STOP command, just as Ctrl-Q restarts screen output previously stopped with Ctrl-S. Just as Ctrl-Q starts to display additional files on the screen if the user has typed file-display commands, CMD_START starts the processing of queued I/O requests.

## *CMD_STOP*

### Purpose of Command

The CMD_STOP command stops command execution immediately. It also prevents the Input device routines from executing any queued command I/O requests. Once unit 0 is stopped by CMD_STOP, the system automatically queues command I/O requests dispatched to unit 0 until CMD_START restarts it.

CMD_STOP is always executed as an immediate-mode command. The results of command execution are found in io_Error, where 0 indicates that the command was successful. IOERR_NOCMD indicates that the task specified the io_Command parameter incorrectly. IOERR_ABORTED indicates that the specified command was aborted with AbortIO or CMD_FLUSH.

### 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 Unit structures that manage Input device unit 0. These can always be copied from the IOStdReq structure initialized by the OpenDevice function call. Also initialize io_Command to CMD-_STOP and set io_Flags to 0.

### Discussion

The CMD_STOP command stops the execution of a command. It is similar to the Ctrl-S command used for screen output on most computers; it stops an executing unit 0 I/O request commands at the earliest possible opportunity.

## DEVICE-SPECIFIC COMMANDS

## *IND_ADDHANDLER*

### Purpose of Command

The IND_ADDHANDLER command adds a new input-handler function to the list of input-handler functions that preprocess input events before they are sent to task-specific routines for further processing. The results of command execution are found in io_Error; 0 indicates that the command was successful. IOERR_NOCMD indicates that the task specified the io_Command parameter incorrectly. IOERR_ABORTED indicates that the specified command was aborted with AbortIO or CMD_FLUSH.

### 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 Unit structures that manage Input device unit 0. These can always be copied from the IOStdReq structure initialized by the OpenDevice function call. Also set io_Command to IND_ADDHANDLER and set io_Flags to 0.

Initialize io_Data to point to an Interrupt structure representing the new input-handler function you want to add to the input-handler function list. Design the Interrupt structure so that its is_Code parameter points to the preprocessing code for the new input handler and its is_Data parameter points to the input event data used by the preprocessing code. The is_Data parameter is the same as the handlerData parameter in the HandlerFunction function call.

### Discussion

IND_ADDHANDLER adds a new input-handler function to the system input-handler function list. Once added, the new function is called by a task in the following manner:

**newInputEvent = HandlerFunction (oldInputEvent, handlerData)**

HandlerFunction is the name of the new input-handler function and provides its entry point; oldInputEvent points to an InputEvent structure representing the first input event to be preprocessed by the new function; and handlerData points to the data used by the new function (note that this is the same as the Interrupt structure is_Data parameter). The newInputEvent pointer points to an InputEvent structure representing the first input event to be preprocessed by the next input-handler function in the list. The HandlerFunction function should be designed so that the newInputEvent parameter is returned when it has finished preprocessing all of its input events.

When a specific HandlerFunction function returns a null value, it indicates that all input events in the input event chain have been preprocessed by it. Lower-priority input-handler functions can preprocess the remaining input events in the input event stream.

## *IND_REMHANDLER*

### Purpose of Command

The IND_REMHANDLER command removes an input-handler function from the input-handler function list. The removed function was previously added to the function list with IND_ADDHANDLER and was used to preprocess input events before they were passed to other task routines; once removed, the function can no longer preprocess input events.

The results of command execution are found in io_Error, where 0 indicates that the command was successful. IOERR_NOCMD indicates that the task specified the io_Command parameter incorrectly. IOERR_ABORTED indicates that the specified command was aborted with AbortIO or CMD_FLUSH.

### 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 Unit structures that manage Input device unit 0. These can always be copied from the IOStdReq structure initialized by the OpenDevice function call. Also initialize io_Command to IND_REMHANDLER and set io_Flags to 0.

Initialize io_Data to point to an Interrupt structure representing the input handler you want to remove from the system. This Interrupt structure was originally designed with the is_Code parameter pointing to the interrupt code for the new function and the is_Data parameter pointing to the data used by the input-handler function.

### Discussion

IND_REMHANDLER removes an input-handler function from the input-handler function list. Both IND_ADDHANDLER and IND_REMHANDLER work with the Interrupt structure to represent the code and data required to define the input-handler function. Once removed, that particular function cannot be used to preprocess input events.

## IND_SETMPORT

### Purpose of Command

The IND_SETMPORT command allows a task to specify the gameport to which the mouse is currently connected. The IOStdReq structure io_Data parameter is used to point to a byte stored in RAM that tells the task which gameport has the mouse. If the byte is 0, the mouse is connected to the left gameport connector; if it is 1, the mouse is connected to the right gameport connector.

The results of command execution are found in io_Error, where 0 indicates that the command was successful. IOERR_NOCMD indicates that the task specified the io_Command parameter incorrectly, and IOERR_ABORTED indicates that the specified command was aborted with AbortIO or CMD_FLUSH. IOERR_BADLENGTH indicates that the task specified the io_Length parameter 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 Device and Unit structures that manage Input device unit 0. These can always be copied from the IOStdReq structure initialized by the OpenDevice function call. Also set io_Command to IND_SETMPORT.

Initialize io_Flags to IOF_QUICK for QuickIO; otherwise, set it to 0. Initialize io_Length to a value of 1. Initialize io_Data to point to a byte value stored in RAM—0 if mouse inputs are to be obtained from the left (front) connector or 1 if they are to be obtained from the right (rear) gameport connector.

### Discussion

IND_SETMPORT allows a task to specify which gameport connector the mouse is connected to. The mouse is usually connected to the left gameport connector, which is nearest to the front of the Amiga. However, because of other hardware requirements in the system, you may want to connect the mouse to the right gameport connector. If the mouse is physically moved, any program depending on the mouse must look for mouse input from the new location.

Your program could also instruct the user to plug the mouse into another connector in order to accommodate other hardware requirements of the system—it can, for example, display an Intuition requester that tells the user to plug the mouse into another gameport connector. At the same time it can send the IND_SETMPORT command to the system so that it will look for mouse input from the new location.

## IND_SETMTRIG

### Purpose of Command

The IND_SETMTRIG command establishes the mouse input trigger conditions that must occur before a pending Gameport device ReadEvent command can be satisfied. The results of command execution are found in io_Error, where 0 indicates that the command was successful. IOERR_NOCMD indicates that the task specified the io_Command parameter incorrectly. IOERR_ABORTED indicates that the specified command was aborted with AbortIO or CMD_FLUSH, and IOERR_BADLENGTH indicates that the task specified the io_Length parameter 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 Device and Unit structures that manage Input device unit 0. These can always be copied from the IOStdReq structure initialized by the OpenDevice function call. Set io_Command to IND_SETMTRIG.

Set io_Flags to IOF_QUICK for QuickIO; otherwise, set it to 0. Set io_Length to the size of the GameportTrigger structure; a task can use the C language sizeof operator to initialize this parameter. Set io_Data to point to a GameportTrigger structure.

### Discussion

IND_SETMTRIG sets the trigger conditions to which the Gameport device ReadEvent command refers in order to determine if mouse movement parameters have changed sufficiently to cause a valid input event. Further details on the ReadEvent command and the GameportTrigger structure are presented in Chapter 10, which discusses the Gameport device.

## IND_SETMTYPE

### Purpose of Command

The IND_SETMTYPE command allows a task to establish the type of device at the mouse connector so that gameport signals coming from that connector may be properly interpreted by the Amiga hardware and the mouse management task. The IOStdReq structure io_Data parameter points to a byte stored in RAM, which tells the task the type of device currently connected to the mouse connector.

The results of command execution are found in io_Error; 0 indicates that the command was successful. IOERR_NOCMD indicates that the task specified the io_Command parameter incorrectly. IOERR_ABORTED indicates that the specified command was aborted with Abort_IO or CMD_FLUSH, and IOERR_BADLENGTH indicates that the task specified the io_Length parameter 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 Device and Unit structures that manage Input device unit 0. These can always be copied from the IOStdReq structure initialized by the OpenDevice function call. Also initialize io_Command to IND_SETMTYPE.

Initialize io_Flags to IOF_QUICK if you want QuickIO; otherwise, set it to 0. Initialize io_Length to a value of 1. Set io_Data to point to a byte value stored in RAM. The meanings of the byte values are as follows:

- If the byte is 0, the connected device is currently allocated to another task.
- If the byte is 1, the connected device is not a controller type that the Amiga understands.
- If the byte is 2, the connected device is a normal Amiga mouse device.
- If the byte is 3, the connected device is a relative joystick device.
- If the byte is 4, the connected device is an absolute joystick device.

## Discussion

IND_SETMTYPE tells the task which type of device is currently connected to the mouse connector, allowing a task and the user to interact when a device is changed at that connector. A task can build an Intuition requester to ask the user to select the type of device installed at the connector. The task can then take this user input and use IND_SETMTYPE to set the device type. Then, later in the program, that task or another can ask the user to select (or verify) the type of device connected. If the device has changed, the current task can once again use IND_SETMTYPE to change the current device type.

The types of devices are defined in the Gameport.h INCLUDE file. The values used for these parameters are GPCT_ALLOCATED, GPCT_NOCONTROLLER, GPCT_MOUSE, GPCT_RELJOYSTICK, and GPCT_ABSJOYSTICK; they correspond to the byte values 0–4. See Chapter 10 for more on these values.

## IND_SETPERIOD

### Purpose of Command

The IND_SETPERIOD command establishes the time interval at which a key repeats its input. If a key is held down longer than the time period established by IND_SETTHRESH, a new keyboard input event is sent to the Input device internal routines. Additional input events are sent at the time interval established by IND_SETPERIOD. This time period applies to all keyboard keys and stays in effect until it is changed by another IND_SETPERIOD command.

IND_SETPERIOD always operates as an immediate-mode command. The results of command execution are found in io_Error, where 0 indicates that the command was successful. IOERR_NOCMD indicates that the task specified the io_Command parameter incorrectly. IOERR_ABORTED indicates that the specified command was aborted with AbortIO or CMD_FLUSH.

### Preparation of the TimeRequest 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 Unit structures that manage Input device unit 0. These can always be copied from the TimeRequest structure initialized by the OpenDevice function call. Set io_Command to IND_SETPERIOD and set io_Flags to 0.

Also initialize the following command-specific parameters:

- tv_Secs. Initialize this to the first component of the key repeat period. This component is measured in seconds; it can be modified from the Intuition Preferences screen.
- tv_Micros. Initialize this to the second component of the key repeat period. This component is measured in microseconds; it too can be modified from the Intuition Preferences screen.

### Discussion

The IND_SETPERIOD command enables a task to specify how much time will elapse between key input events to the Input device internal routines. For example, if a task wants to ignore keyboard input events temporarily, it can set tv_Secs and tv_Micros to values that represent a very long interval. Then, no matter how long the user holds down a key, the Input device internal routines will not receive another keyboard input event. If a task wanted the keyboard keys to repeat very quickly, it could set tv_Secs to 0 and tv_Micros to a very small value.

The IND_SETPERIOD and IND_SETTHRESH commands allow a task to specify how it will deal with keyboard input events from the user. These commands are usually dispatched by Intuition, and their default values are represented by TimeVal substructures in the Intuition Preferences structure; they can be indirectly executed by making choices from the Intuition Preferences screen. Both commands use the TimeRequest structure to specify their I/O requests to the Input device internal routines.

## *IND_SETTHRESH*

### Purpose of Command

The IND_SETTHRESH command establishes the threshhold time period after which a keyboard key will be considered to repeat its input for the first time. This value stays in effect until changed by another IND_SETTHRESH command. The time period applies to all keyboard keys. IND_SETTHRESH always operates as an immediate-mode command, and the results of command execution are found in io_Error. A 0 value indicates that the command was successful. IOERR_NOCMD indicates that the task specified the io_Command parameter incorrectly, and IOERR_ABORTED indicates that the specified command was aborted with AbortIO or CMD_FLUSH.

### 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 Unit structures that manage Input device unit 0. These can always be copied from the TimeRequest structure initialized by the OpenDevice function call. Set io_Command to IND_SETTHRESH, and set io_Flags to 0. Also, initialize the following command-specific parameters:

- tv_Secs. Initialize this to the key repeat period. This component is measured in seconds; it can be modified from the Intuition Preferences screen.
- tv_Micros. Initialize this to the key repeat period. This component is measured in microseconds; it too can be modified from the Intuition Preferences screen.

### Discussion

IND_SETTHRESH allows a task to establish a time-period threshhold after which a key that is continuously held down will be considered to repeat its input, thus producing another keyboard input event. When the key is held down for longer than this threshhold, the key repeat values established by the last IND_SETPERIOD command determine the time period between subsequent keyboard input events.

IND_SETPERIOD and IND_SETTHRESH are usually issued by Intuition. Their default values are represented as TimeVal substructures in the Intuition Preferences structure and can be indirectly executed from the Intuition Preferences screen. Both commands use the TimeRequest structure to specify their I/O requests to the Input device routines.

## IND_WRITEEVENT

### Purpose of Command

The IND_WRITEEVENT command allows a task to define its own input events and to add them to the input event stream, which will then consist of a set of task-defined input events interspersed with Keyboard, Gameport, and Timer device input events, as well as disk insertion and removal events. IND_WRITEEVENT uses the IOStdReq structure io_Length and io_Data parameters to specify the RAM location of the InputEvent structures representing a series of input events. The entire buffer is io_Length bytes in length; its size is determined by the number of InputEvent structures.

IND_WRITEEVENT is always treated as a synchronous I/O request and always replies to the task reply-port queue if the IOF_QUICK bit is not set. The results of command execution are found in io_Error, where 0 indicates that the command was successful. IOERR_NOCMD indicates that the task specified the io_Command parameter incorrectly, and IOERR_ABORTED indicates that the specified command was aborted with AbortIO or CMD_FLUSH. IOERR_BADLENGTH indicates that the task specified the io_Length parameter 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 Device and Unit structures that manage Input device unit 0. These can always be copied from the IOStdReq structure initialized by the OpenDevice function call. Set io_Command to IND_WRITE-EVENT; set io_Flags to 0 or to IOF_QUICK for QuickIO. Also, initialize the following command-specific parameters:

- io_Length. Initialize this to the size of the input event buffer, which is the number of InputEvent structures times the size of each InputEvent structure. A task can used the C language sizeof operator to determine the number of bytes in the buffer.

- io_Data. Initialize this to point to a RAM buffer area that will contain a series of InputEvent structures representing the new input events that will be passed to the Input device for processing.

## Discussion

The IND_WRITEEVENT command allows a task to add its own input events to an event stream. You may find this useful to simulate the keyboard, the timer, or a mouse without dealing directly with their input events; you can also use this command to debug a complicated program that deals with these devices. In other cases, you may find it useful to construct a set of input events for a number of uses in one or more tasks.

# The Console Device

8

## 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

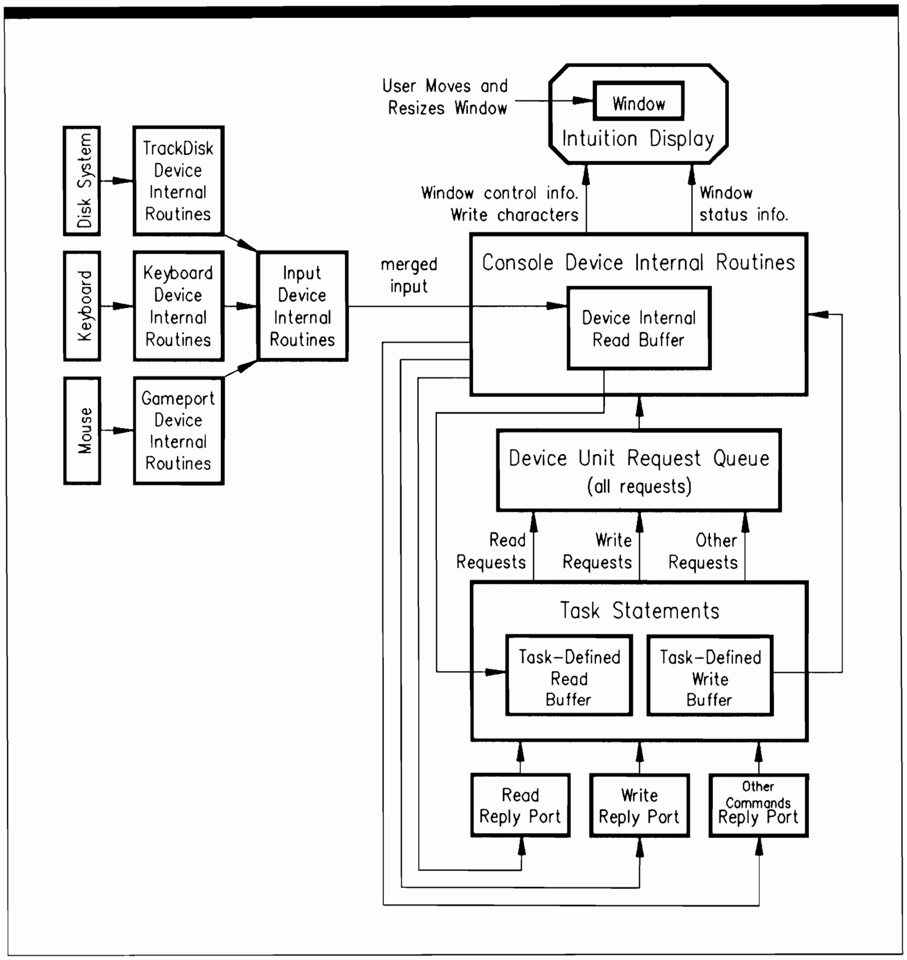

User Moves and Resizes Window
Window
Intuition Display
Window control info. Write characters
Window status info.
Disk System
TrackDisk Device Internal Routines
Keyboard
Keyboard Device Internal Routines
Input Device Internal Routines
Mouse
Gameport Device Internal Routines
merged input
Console Device Internal Routines
Device Internal Read Buffer
Device Unit Request Queue (all requests)
Read Requests
Write Requests
Other Requests
Task Statements
Task-Defined Read Buffer
Task-Defined Write Buffer
Read Reply Port
Write Reply Port
Other Commands Reply Port

**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.

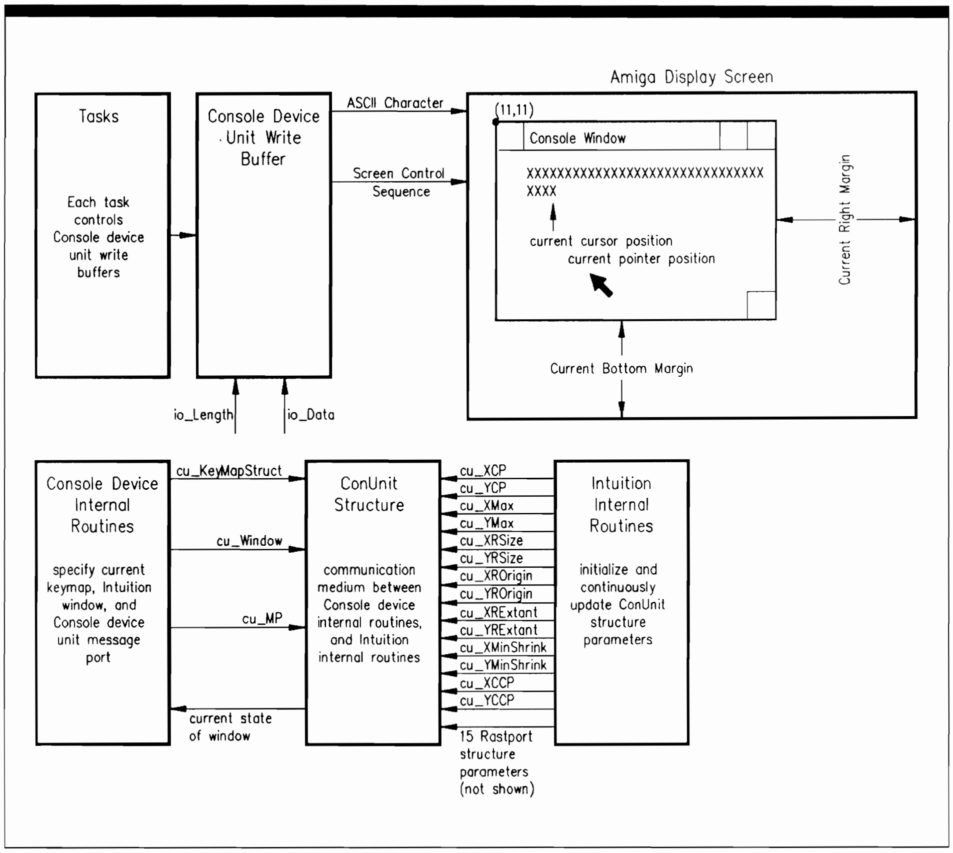

Tasks
Each task controls Console device unit write buffers
Console Device Unit Write Buffer
ASCII Character
Screen Control Sequence
Amiga Display Screen
(11,11)
Console Window
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXX
current cursor position
current pointer position
Current Right Margin
Current Bottom Margin
io_Length
io_Data
Console Device Internal Routines
specify current keymap, Intuition window, and Console device unit message port
cu_KeyMapStruct
cu_Window
cu_MP
current state of window
ConUnit Structure
communication medium between Console device internal routines, and Intuition internal routines
cu_XCP
cu_YCP
cu_XMax
cu_YMax
cu_XRSize
cu_YRSize
cu_XROrigin
cu_YROrigin
cu_XRExtant
cu_YRExtant
cu_XMinShrink
cu_YMinShrink
cu_XCCP
cu_YCCP
15 Rastport structure parameters (not shown)
Intuition Internal Routines
initialize and continuously update ConUnit structure parameters

**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_SETKEYMAP functions to manage the key map.

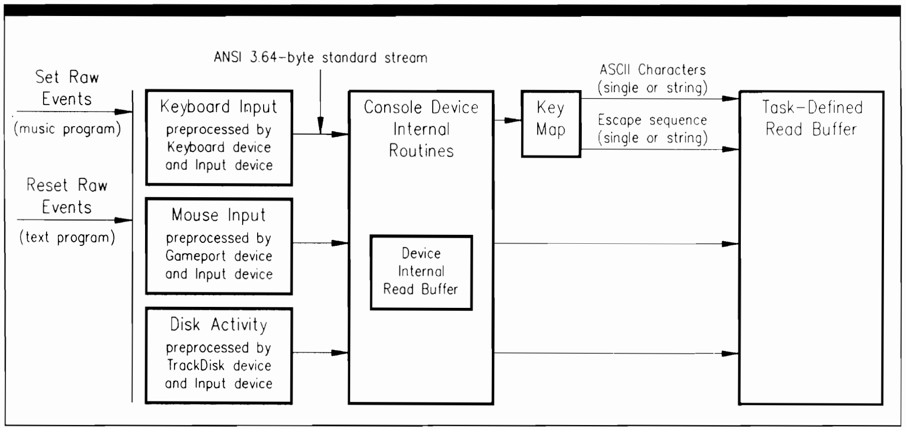

ANSI 3.64-byte standard stream
Set Raw Events
(music program)
Reset Raw Events
(text program)
Keyboard Input preprocessed by Keyboard device and Input device
Mouse Input preprocessed by Gameport device and Input device
Disk Activity preprocessed by TrackDisk device and Input device
Console Device Internal Routines
Device Internal Read Buffer
Key Map
ASCII Characters (single or string)
Escape sequence (single or string)
Task-Defined Read Buffer

**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 character 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 QuickIO 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.

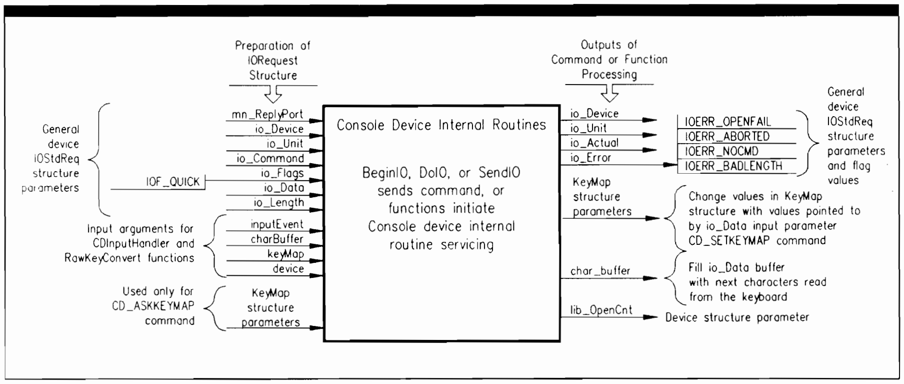

Preparation of IORequest Structure
Outputs of Command or Function Processing
General device IOStdReq structure parameters
IOF_QUICK
mn_ReplyPort
io_Device
io_Unit
io_Command
io_Flags
io_Data
io_Length
Input arguments for CDInputHandler and RawKeyConvert functions
inputEvent
charBuffer
keyMap
device
Used only for CD_ASKKEYMAP command
KeyMap structure parameters
Console Device Internal Routines
BeginIO, DoIO, or SendIO sends command, or functions initiate Console device internal routine servicing
io_Device
io_Unit
io_Actual
io_Error
IOERR_OPENFAIL
IOERR_ABORTED
IOERR_NOCMD
IOERR_BADLENGTH
General device IOStdReq structure parameters and flag values
KeyMap structure parameters
Change values in KeyMap structure with values pointed to by io_Data input parameter CD_SETKEYMAP command
char_buffer
Fill io_Data buffer with next characters read from the keyboard
lib_OpenCnt
Device structure parameter

**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.

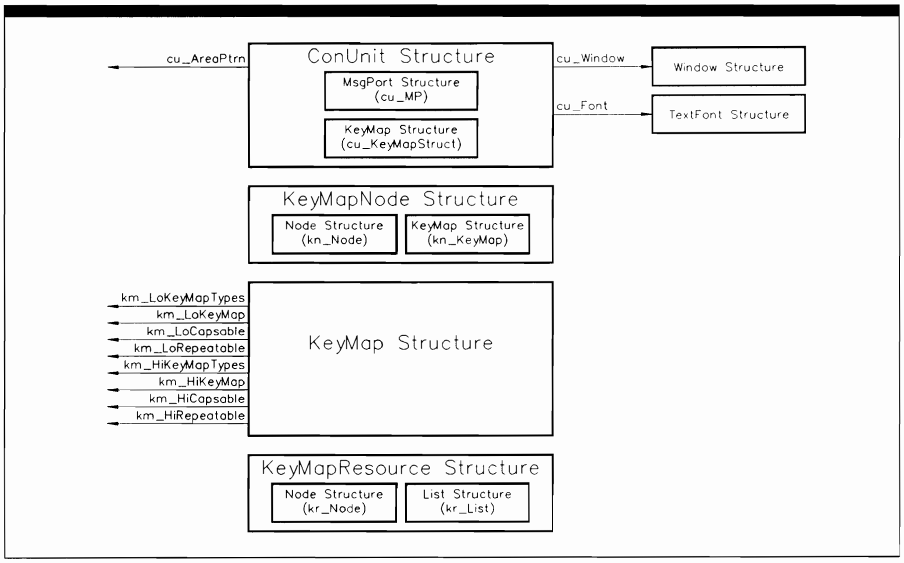


**Figure 8.5:** *Console Device Structures*

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 kr_List. 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 ConUnit {
   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)/8];
    };
```

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 ConUnit 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 ConUnit 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_HiRepeatable 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, 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,
D0                        A0          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_Actual parameter, which contains the actual number of ANSI byte characters 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 Key-Map 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 QuickIO 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

### Purpose 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_ASKKEYMAP and CD_SETKEYMAP commands.

CMD_READ 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_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 1. A 0 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 QuickIO, 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 F1 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 QuickIO 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 QuickIO, 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.

## Discussion

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_BADLENGTH 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_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_ASKDEFAULTKEYMAP. Also initialize the following command-specific parameters:

- io_Flags. Set this to 0 if not used. Otherwise, set it to IOF_QUICK for QuickIO; QuickIO 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 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 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 QuickIO; otherwise, set it to 0. QuickIO 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_ASKKEYMAP 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 QuickIO and only replies to the task reply-port queue if QuickIO 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 QuickIO, 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 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; 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 QuickIO, 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.

# 9 The Keyboard Device

## Introduction

The Keyboard device is responsible for collecting information as the user enters it from the keyboard. It operates in shared access mode and has one unit, referred to as unit 0. The Keyboard device is ROM-resident; it is automatically loaded into WCS from the Kickstart disk.

Information coming from the keyboard includes raw key codes, the current up or down status of any key, the current status of the left and right Shift keys, and the status of numeric keypad keys. The Keyboard device receives this raw information and converts it into a set of input events. Just like other Amiga input events, these are represented as InputEvent structures constructed by the device. (The InputEvent structure is discussed in Chapter 7.)

The Keyboard device also buffers Amiga keystrokes by placing a series of keyboard InputEvent structures into its internal read buffer, which provides a typeahead capability for the keyboard. This buffer is managed by the Keyboard device internal routines automatically; do not confuse it with the task-defined buffer used by the KBD_READEVENT command.

Once keyboard input events are in the Keyboard device internal read buffer, they can be processed in several different ways. AmigaDOS, if active, will process reset events, and a task can process input events as the programmer desires. However, if AmigaDOS or a task does not process the events, the Input device will receive them and they will be merged with those coming from the Gameport and TrackDisk devices, as well as from any other sources in the Amiga system.

The Keyboard device allows a task to determine the current status (up or down) of each Amiga keyboard key with the KBD_READMATRIX command. It also provides the KBD_ADDRESETHANDLER, KBD_REMRESETHANDLER, and KBD_RESETHANDLERDONE commands so that a task can add reset-handler functions to the system. These functions are placed on a function list and are processed in a specific order, which allows the programmer to design a series of priority-arranged cleanup routines that can be called each time the user types the Ctrl/left-Amiga/right-Amiga reset combination.

## Operation of the Keyboard Device

Figure 9.1 illustrates the general operation of the Keyboard device. Keyboard data originates at the keyboard, is placed into an internal read buffer by the Keyboard device internal routines, and is passed on to the Input device internal routines for further processing.

As the figure shows, a task can build a number of keyboard reset-handler functions into the keyboard system. These are placed on a keyboard reset-handler list in the priority established by each reset handler's ln_Pri parameter. Then, when the user presses the Amiga reset key combination, the reset-handler functions will implement the reset procedures built into the system. When all of the functions have executed, the Amiga will go through its own reset sequence, which is defined in the system software.

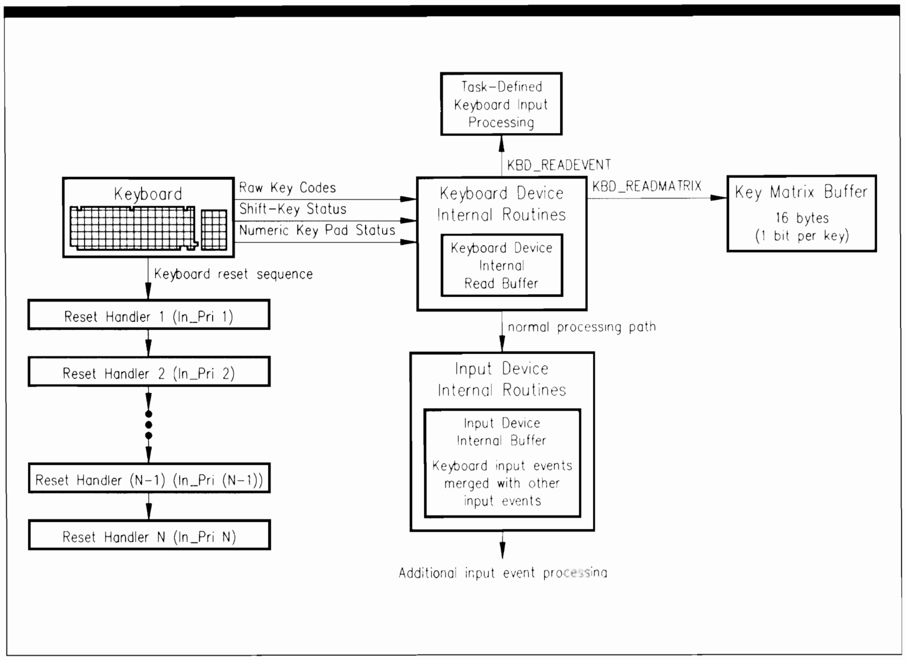

Task-Defined Keyboard Input Processing
KBD_READEVENT
Keyboard
Raw Key Codes
Shift-Key Status
Numeric Key Pad Status
Keyboard Device Internal Routines
Keyboard Device Internal Read Buffer
KBD_READMATRIX
Key Matrix Buffer
16 bytes (1 bit per key)
Keyboard reset sequence
Reset Handler 1 (In_Pri 1)
Reset Handler 2 (In_Pri 2)
Reset Handler (N-1) (In_Pri (N-1))
Reset Handler N (In_Pri N)
normal processing path
Input Device Internal Routines
Input Device Internal Buffer
Keyboard input events merged with other input events
Additional input event processing

**Figure 9.1:** *Operation of the Keyboard Device*

Notice from Figure 9.1 that the Amiga keyboard provides N-key typeahead, which allows the user to type at his or her own speed. No keys will be lost, even though the system may be busy with other activities.

## Keyboard Input Event Processing

All keyboard input events are processed by a predefined sequence of routines, as shown in Figure 9.2. The events originate in the keyboard hardware as signals, which are passed to the Keyboard device internal routines for processing.

The internal routines formulate events into InputEvent structures and place them into the Keyboard device internal read buffer. Event processing then follows this sequence:

1. If AmigaDOS is active—that is, if the user is at the AmigaDOS command level—the AmigaDOS internal routines will read the current contents of the Keyboard device internal read buffer and filter out any keyboard input events that they have been programmed to intercept. In particular, the AmigaDOS routines will always intercept the Amiga reset sequence. (If a program has control of the machine, however, the AmigaDOS routines will not intercept keyboard input events.)

2. Keyboard input events that are not intercepted by AmigaDOS routines are passed on for further processing. At this point, a task can intercept the keyboard input events and place them into its own read buffer using KBD_READEVENT. It can then define a set of task routines to process the events. This procedure processes keyboard

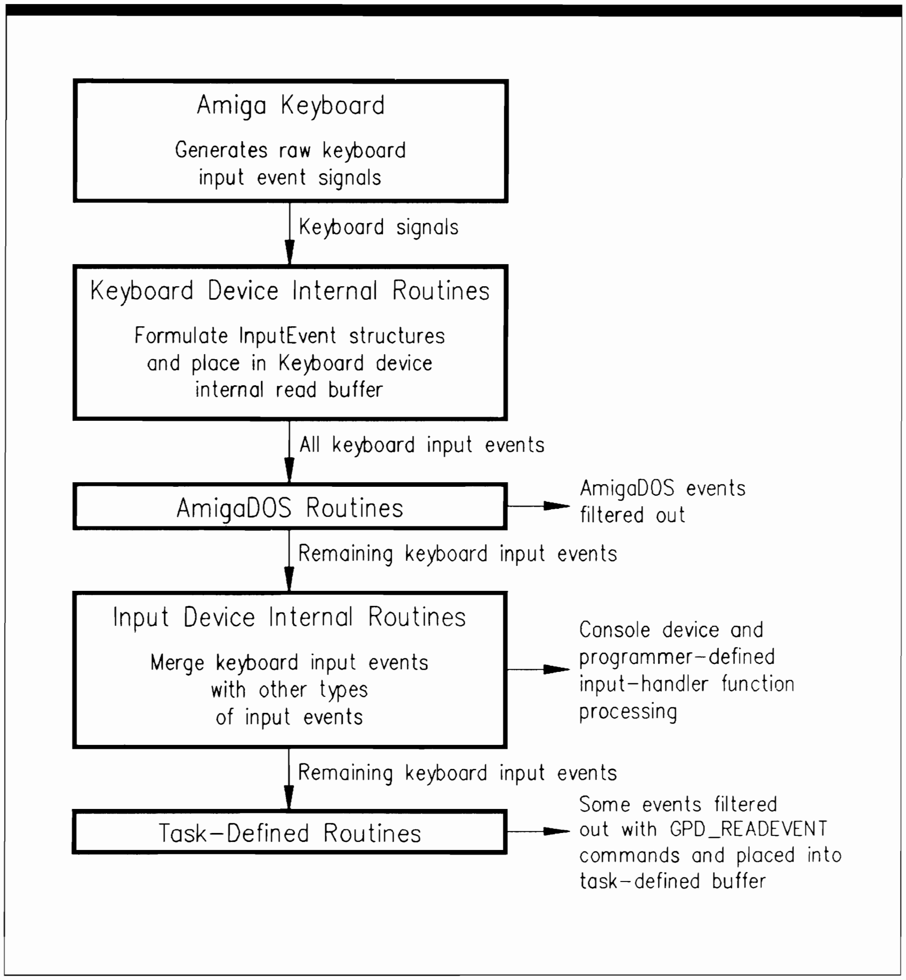

Amiga Keyboard
Generates raw keyboard input event signals
Keyboard signals
Keyboard Device Internal Routines
Formulate InputEvent structures and place in Keyboard device internal read buffer
All keyboard input events
AmigaDOS Routines
AmigaDOS events filtered out
Remaining keyboard input events
Input Device Internal Routines
Merge keyboard input events with other types of input events
Console device and programmer-defined input-handler function processing
Remaining keyboard input events
Task-Defined Routines
Some events filtered out with GPD_READEVENT commands and placed into task-defined buffer

**Figure 9.2:** *Input Event Processing for the Keyboard Device*

input events, but it allows the programmer to choose how to process them. For example, a task could place the current key matrix into a task-defined buffer using the KBD_READMATRIX command. This would allow it to read the current state (up or down) of the keyboard keys. The key matrix represents the state of each keyboard key by one bit in a 16-byte array; the task checks each bit in the key matrix to determine its next action.

3. Keyboard input events that are not intercepted by either AmigaDOS or a task are passed on to the Input device internal routines for further processing. Input device routines merge the Keyboard device input events with Gameport and TrackDisk device input events, as well as any task-defined input events (see Chapter 7). The

Input device internal routines then pass all events in this merged stream to the Console device, which will intercept the Amiga reset sequence.

## Keyboard Device Commands

You can program the Keyboard device with five device-specific and two standard device commands. Some of these commands support both QuickIO and queued I/O; none supports immediate-mode operation. All commands affect the IOStdReq structure io_Error parameter; CMD_CLEAR also affects the contents of the Keyboard device internal read buffer.

### Sending Commands to the Keyboard Device

Figure 9.3 depicts the general scheme used to send commands to the Keyboard device internal routines. The lines with arrows represent the parameters you should initialize and the parameters returned by the device internal routines.

The Keyboard device programming process consists of three phases:

1. Structure preparation. The programmer has complete control over this phase. Here, you initialize parameters in the IOStdReq structure in preparation for sending a command to the Keyboard device internal routines. These parameters include the set of parameters required by most devices; in addition, the KBD_ADDRESET-HANDLER command requires the is_Data and is_Code pointer parameters to define an Interrupt structure representing a keyboard reset-handler function. The choice of parameters to initialize depends on the specific command you plan to send to the Keyboard device. These parameters provide an information path to the data needed by the Keyboard device internal routines in order to process the command.

2. Keyboard device processing. The only part you play in this phase is to send the command to the device using the BeginIO, DoIO, or SendIO function. Control then passes to the device and system internal routines.

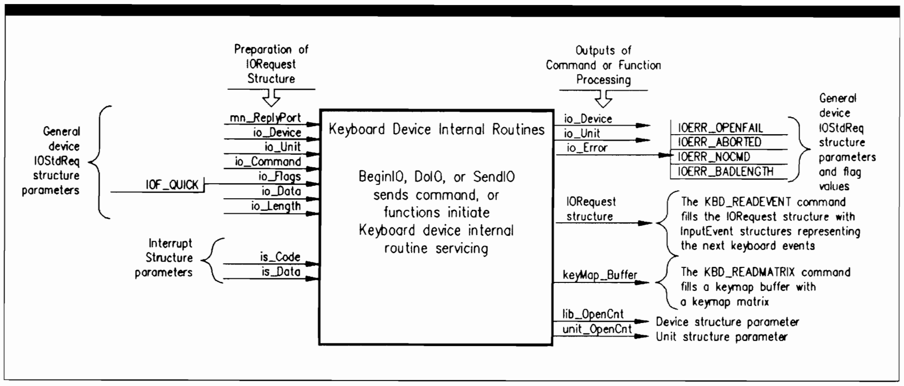
3. Command output parameter processing. The system and Keyboard device internal routines have complete control over the values found in these parameters. Here, the results of Keyboard device command processing have been returned to the task that originally dispatched the command. If the I/O request was not QuickIO, it was processed when it moved to the top of the Keyboard device request queue; the reply is now in the task reply-port queue. If the request was QuickIO, it was not queued in the task reply-port queue but went directly to the requesting task. The parameters still direct you to the appropriate data for your task.

All Keyboard device commands provide output parameters; in each case, the IOStdReq structure io_Error parameter is the only parameter provided. In addition, the KBD_READ-EVENT command fills a task-defined buffer with InputEvent structures for the next set of keyboard input events; KBD_READMATRIX fills a task-defined buffer with the current key matrix.

Figure 9.3 also depicts the parameters that play a part in Keyboard device function setup and processing. The OpenDevice and CloseDevice functions affect the Unit structure unit_OpenCnt parameter and the Device structure lib_OpenCnt parameter; OpenDevice also affects the io_Error parameter.

## Structures for the Keyboard Device

The Keyboard device does not have any device-specific structures that are specifically tied to its commands or functions. However, you should refer to Chapter 8 for a discussion of the KeyMap, KeyMapNode, and KeyMapResource structures, which also have a bearing on the operations of the Keyboard device.



Preparation of IORequest Structure
Outputs of Command or Function Processing
General device IOStdReq structure parameters
mn_ReplyPort
io_Device
io_Unit
io_Command
IOF_QUICK
io_Flags
io_Data
io_Length
Interrupt Structure parameters
is_Code
is_Data
Keyboard Device Internal Routines
BeginIO, DoIO, or SendIO sends command, or functions initiate Keyboard device internal routine servicing
io_Device
io_Unit
io_Error
IOERR_OPENFAIL
IOERR_ABORTED
IOERR_NOCMD
IOERR_BADLENGTH
General device IOStdReq structure parameters and flag values
IORequest structure
The KBD_READEVENT command fills the IORequest structure with InputEvent structures representing the next keyboard events
keyMap_Buffer
The KBD_READMATRIX command fills a keymap buffer with a keymap matrix
lib_OpenCnt
unit_OpenCnt
Device structure parameter
Unit structure parameter

**Figure 9.3:** *Keyboard Device Command and Function Processing*

## USE OF FUNCTIONS

## *CloseDevice*

## Syntax of Function Call

**CloseDevice (iOStdReq)**
**A1**

## Purpose of Function

This function closes access to Keyboard device unit 0. If this is the last CloseDevice function call for unit 0 and the Console and Input devices have also been closed in the task, the Keyboard device will be closed.

CloseDevice sets the IOStdReq structure io_Device and io_Unit parameters to −1; a task cannot use that structure again until these parameters are reinitialized by OpenDevice. CloseDevice also reduces the Device structure lib_OpenCnt and Unit structure unit_OpenCnt parameters by 1 to indicate that one less task is using the Keyboard device.

### Inputs to Function

|  |  |
|---|---|
| **iOStdReq** | A pointer to an IOStdReq structure |

### Discussion

CloseDevice terminates access to a set of device routines for Keyboard device unit 0. When a task is done with its Keyboard device operations, it should close the device with a call to CloseDevice; this frees memory that might be needed by the system. Another task that wants to use the Keyboard device can then open, use, and close it. This sequence can be repeated again and again in a C language program that uses the Keyboard device routines.

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

Always remember that the Keyboard device is opened indirectly when the Input or Console device is opened; it is opened automatically by AmigaDOS as well. Therefore, the particular procedure used to open the Keyboard device should determine how you proceed with your CloseDevice calls.

## *OpenDevice*

### Syntax of Function Call

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

### Purpose of Function

OpenDevice allows access to the internal routines of Keyboard device unit 0. The Keyboard device is a shared access mode device, and it is opened automatically by either AmigaDOS or the Console and Input devices when they are opened.

Once OpenDevice has opened the Keyboard device, it then initializes Keyboard device internal parameters to their most recently specified or default values. OpenDevice also increments the Device structure lib_OpenCnt parameter and the Unit structure unit_OpenCnt parameter by 1 to indicate that one more task is using the Keyboard device.

OpenDevice requires a properly initialized reply port with a task signal bit allocated to that port if the calling task wants to be signaled when the Keyboard device internal routines reply. The task can then be signaled when any of the Keyboard device commands are replied. The results of function execution are as follows:

- io_Device. This points to a Device structure that will manage Keyboard device unit 0 once it is opened. The Device structure contains all the information necessary to manage the Keyboard device and to reach all the data and routines in it.
- io_Unit. This points to a Unit structure that defines and manages a MsgPort structure for Keyboard device unit 0. The MsgPort structure represents the device request queue. Because the Keyboard device operates in shared access mode, tasks send their non-QuickIO requests to this message port.
- io_Error. A 0 value indicates that the requested open succeeded. IOERR_OPENFAIL indicates that the Keyboard device could not be opened; this usually means that the memory required to accommodate the Device and Unit structures was not available. IOERR_NOCMD indicates that the io_Command parameter was specified incorrectly.

## Inputs to Function

| | |
|---|---|
| **"keyboard.device"** | A pointer to a null-terminated string representing the name of the Keyboard device |
| **0** | The only valid unit number |
| **iOStdReq** | A pointer to an IOStdReq structure |
| **0** | Indicates that the flags argument is not used by this device |

## Preparation of the IOStdReq Structure

Initialize mn_ReplyPort to point to a MsgPort structure for the task reply port, which will receive the I/O request reply from the Keyboard device when it has finished processing the Keyboard device command. Set all other parameters in the IOStdReq substructure to 0, or copy them from an IOStdReq structure from a previous OpenDevice call. Initialize io_Command to 0, or set it to KBD_READEVENT if the task should open the Keyboard device and then dispatch a KBD_READEVENT command.

If the CreateStdIO function is used to create the IOStdReq structure (see Chapter 2), CreateStdIO will automatically return a pointer to an IOStdReq structure; for the Keyboard device, no typecasting will be necessary.

## Discussion

The OpenDevice function allows a task to access the Keyboard device internal routines. One of the things that makes the Keyboard device unique is that it is opened automatically by AmigaDOS or by the Console or Input device when an OpenDevice call for these two devices is executed.

The Keyboard device internal parameters will be set to default values the first time the Keyboard device is opened in a task. Once the Keyboard device has been opened, other IOStdReq structure parameters can be initialized to define I/O requests for Keyboard device reads and other commands. Any parameters that are not explicitly initialized will retain their previous values. If a calling task wants to use values other than the defaults for these parameters, it should initialize them after OpenDevice returns.

OpenDevice is usually called with appropriate parameters to open the Keyboard device and to initialize parameters to define a KBD_READEVENT command. Once a task has opened the Keyboard device, it can dispatch a series of KBD_READEVENT and device-specific commands (with BeginIO, DoIO, or SendIO) to gather information from the Amiga keyboard and to send it to the task, the Input device, or the Console device for further processing. Once a task has finished its Keyboard device processing, it should close the Keyboard device.

## STANDARD DEVICE COMMANDS

## *CMD_CLEAR*

### Purpose of Command

CMD_CLEAR clears the Keyboard device internal read buffer, which is an internal device buffer used only to save typeahead keystrokes (InputEvent structures) to be read later by the KBD_READEVENT command. Once the read buffer is cleared, subsequent KBD_READEVENT commands can proceed from a known buffer state.

The CMD_CLEAR command allows QuickIO and always replies to the task reply-port queue if QuickIO 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 command was aborted, and IOERR_NOCMD indicates that the io_Command 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 Device and Unit structures that manage unit 0 of the Keyboard device. These can always be copied from the IOStdReq structure initialized by the OpenDevice function call. Also initialize io_Command to CMD_CLEAR. Initialize io_Flags to 0, or set it to IOF_QUICK for QuickIO, which may or may not succeed depending on conditions in the system.

### Discussion

The Keyboard device automatically maintains one internal device buffer for KBD_READEVENT commands. The CMD_CLEAR command clears the current contents of this buffer. If a task wants to be sure that the Keyboard device internal read buffer is in a known state before proceeding with a series of KBD_READEVENT commands, it should first dispatch a CMD_CLEAR command to zero all bytes in the buffer. Then subsequent KBD_READEVENT commands will not encounter extraneous characters left over from previously executed KBD_READEVENT commands. The Keyboard device does not have a CMD_WRITE command and therefore uses no internal device write buffers.

CMD_CLEAR can be either a QuickIO or a queued I/O command. If QuickIO is successful, it acts similarly to an immediate-mode command, except that it is not replied to the task reply-port queue. If queued, the task can execute CMD_CLEAR after a series of already queued KBD_READEVENT commands, which allows it to clear the internal read buffer at the proper time.

## *CMD_RESET*

### Purpose of Command

CMD_RESET resets the Keyboard device to the boot-up time state as if it were just initialized—all Keyboard device internal parameters and flag parameter bits are set to their default values. CMD_RESET allows QuickIO and always replies to the task reply-port queue if the QuickIO was not successful. The results of command execution are found in io_Error; 0 indicates that the command was successful. IOERR_ABORTED indicates that the command was aborted, and IOERR_NOCMD indicates that the io_Command 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 Device and Unit structures that manage unit 0 of the Keyboard device. These can always be copied from the IOStdReq structure initialized by the OpenDevice function call. Also initialize io_Command to CMD_RESET. Initialize io_Flags to 0, or set it to IOF_QUICK for QuickIO, which may or may not succeed depending on conditions in the system.

### Discussion

The CMD_RESET command is destructive—it stops all ongoing KBD_READEVENT I/O requests. The dispatching task then loses the keystrokes that it originally requested from the keyboard through the Keyboard device. CMD_RESET also resets the Keyboard device internal parameters and flag parameter bits to their default values.

## DEVICE-SPECIFIC COMMANDS

## *KBD_ADDRESETHANDLER*

### Purpose of Command

The KBD_ADDRESETHANDLER command adds a new keyboard reset handler function to the list of keyboard reset-handler functions that can be called before the machine is reset. The new function is added to the list at the position determined by the function's ln_Pri parameter; the IOStdReq structure io_Data parameter points to an Interrupt structure representing the code and data required to define the new function.

The results of command execution are found in io_Error; 0 indicates that the command was successful. IOERR_ABORTED indicates that the command was aborted, and IOERR_NOCMD indicates that the io_Command 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 Device and Unit structures that manage Keyboard device unit 0. These can always be copied from the IOStdReq structure initialized by the OpenDevice function call. Set io_Command to KBD_ADDRESETHANDLER and set io_Flags to 0. Initialize io_Data to point to an Interrupt structure representing the new function and the handler data you want to add to the list.

Design the Interrupt structure with its is_Code parameter pointing to the keyboard reset code for the new handler and its is_Data parameter pointing to the data used by the new keyboard reset code (the same data as that in the handlerData parameter in the Handler-Function function call).

## Discussion

KBD_ADDRESETHANDLER adds a new keyboard reset-handler function to the system's keyboard reset-handler function list. The new function's position in the list is determined by the task's specification of the Interrupt structure Node substructure ln_Pri parameter representing the new HandlerFunction function.

Once added to the system's keyboard reset-handler function list, the new handler function is called by a task in the following way:

```
HandlerFunction (handlerData)
                      A1
```

HandlerFunction is the name of the new keyboard reset-handler function and its entry point; handlerData is a pointer to the data used by the new function. It is the same as the is_Data parameter in the associated Interrupt structure.

An added reset-handler function will be active when the user resets the system with Ctrl/left-Amiga/right-Amiga. It will be active in the system until KBD_REMRESET-HANDLER removes it from the system's keyboard reset-handler function list.

All reset events can be intercepted by a set of custom-designed keyboard reset-handler functions. User-induced reset events can call each of the functions in the function list in the order of their priority; higher-priority handler functions process the reset events before lower-priority handler functions do. With this arrangement, each reset-handler function can perform a cleanup operation for a task. The cleanup will take place before the machine is actually reset; that is, the reset-handler functions will execute before other operations of the machine reset occur. For example, if the machine is reset while a disk write operation is in progress, a reset-handler function can be designed to protect the contents of the disk by ensuring that the last track buffers are properly written before they are cleared. Each reset-handler function gets its turn at cleaning up.

A task that adds a reset-handler function to the list must design it so that lower-priority functions in the list will get their turn at processing before the machine is finally reset.

## *KBD_READEVENT*

### Purpose of Command

The KBD_READEVENT command allows a task to read the next series of keyboard input events from the Keyboard device internal read buffer. KBD_READEVENT uses the IOStdReq structure io_Length and io_Data parameters to specify the task-defined RAM location where the InputEvent structures will be placed when KBD_READEVENT executes. The required size of the task buffer is determined by the number of InputEvent structures present in it. If there are no pending keyboard input events in the Keyboard device read buffer, the KBD_READEVENT command will not be satisfied. However, if there are some keyboard input events not yet read, but not as many as indicated by the IOStdReq structure io_Length parameter, KBD_READEVENT will be satisfied with only those that are currently available in the buffer.

KBD_READEVENT allows QuickIO and always replies to the task reply-port queue if QuickIO 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 command 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 Device and Unit structures that manage Keyboard device unit 0. These can always be copied from the IOStdReq structure initialized by the OpenDevice function call. Set io_Command to KBD_READEVENT. Initialize io_Flags to IOF_QUICK for QuickIO; otherwise, set it to 0. Initialize io_Length to the size (in bytes) of the desired task-defined keyboard read buffer. To determine the size of the buffer, multiply the number of InputEvent structures by the size of the InputEvent structure; a task can use the C language sizeof operator to determine this value. Initialize io_Data to point to a RAM buffer area that will contain the InputEvent structures.

## Discussion

KBD_READEVENT is responsible for reading keyboard keystrokes into a task-defined buffer from the Keyboard device read buffer. The keystrokes are defined by InputEvent structures. Each InputEvent structure in the task-defined buffer will contain the following parameters:

- ie_NextEvent. This points to the next InputEvent structure in the linked list; it will be 0 if this is the last InputEvent structure in the linked list.
- ie_Class. This is the class of the keyboard event; it is IECLASS_RAWKEY for raw keyboard input events.
- ie_SubClass. This parameter is not used for IECLASS_RAWKEY keyboard input events; the Keyboard device routines set it to 0.

- ie_Code. This contains the next key status report (up or down). Each of the Amiga keyboard keys is assigned a hexadecimal value from 00 to 67. This parameter contains the value assigned for a key-up transition; the value assigned for a key-down transition is a bitwise OR of this value with a hexadecimal value of 80.
- ie_Qualifier. This indicates whether the key was pressed with the left or right Shift key also held down, and whether the key was a numeric keypad key.
- ie_X, ie_Y, and ie_TimeStamp. These parameters are not used for keyboard IECLASS_RAWKEY input events; the Keyboard device routines set them to 0.

Keyboard event reading is not usually done by calling the Keyboard device KBD_READEVENT command directly. Instead, keyboard input events are automatically passed to the Input device. (See Chapter 7 for a discussion of the interaction of the Input and the Keyboard devices.)

Keyboard device internal routines usually queue more than one input event into the Keyboard device read buffer. These events will continue to queue in the buffer until a task dispatches a KBD_READEVENT command or until the Input device automatically reads them. If neither the task nor the input device reads them, the Keyboard device read buffer will overflow; any additional keystrokes beyond those already stored in the buffer will be lost.

## KBD_READMATRIX

### Purpose of Command

The KBD_READMATRIX command is used to read the current key matrix into a task-defined buffer. The key matrix defines the current status (up or down) of every key on the Amiga keyboard. It is automatically updated by the Keyboard device internal routines each time the user presses a key. The IOStdReq structure io_Length and io_Data parameters are used to describe the current key matrix; io_Length defines the number of bytes in the key matrix, and io_Data points to the task-defined buffer that will contain it when the KBD_READMATRIX command finishes execution.

KBD_READMATRIX allows QuickIO and always replies to the task reply-port queue if QuickIO 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 command was aborted, and IOERR_NOCMD indicates that the io_Command parameter was specified incorrectly. 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 Device and Unit structures that manage Keyboard device unit 0; these can always be copied from the IOStdReq structure initialized by the OpenDevice function call. Set io_Command to KBD_READMATRIX. Initialize io_Flags to IOF_QUICK for QuickIO; otherwise, set it to 0. Initialize io_Data to point to the buffer area to be filled with the values in the current key matrix. Set io_Length to the maximum number of bytes in the key matrix. This parameter must be specified as large enough to hold the entire matrix; if it is not, adjacent RAM will be overwritten.

## Discussion

KBD_READMATRIX is the only command that works with the keyboard key matrix. It reads the current values in the key matrix into a task-defined buffer. The Keyboard device internal routines automatically update the key matrix whenever a key is pressed.

The key matrix contains a series of bytes arranged in a predefined sequence. Each byte consists of eight bits, and each bit represents the up or down status of one specific keyboard key. The IOStdReq structure io_Length and io_Data parameters define the number of bytes in the key matrix and its RAM location; they must be initialized before the KBD_READMATRIX command is dispatched. Once a task places the key matrix into a task-defined buffer with KBD_READMATRIX, it can look at each bit to determine the status of keys.

Use the following procedure to design your tasks so they can find the status of a keyboard key:

1. Use to the Amiga keyboard key layout shown in the *ROM Kernel Manual* to determine the bit number corresponding to each keyboard key. For example, the manual shows that the the F2 function key has bit number 51 assigned to it.

2. Find the key matrix bit. For example, for the F2 function key, this is done by first dividing the key matrix bit-position value by 8 (hexadecimal 51 = decimal 81). This indicates that the bit is in byte 10 of the key matrix.

3. Take the same bit position value modulo 8 to determine which bit position within the byte represents the status (up or down) of the keyboard key. A bit value of 0 indicates the key is up; 1 indicates it is down.

## KBD_REMRESETHANDLER

### Purpose of Command

The KBD_REMRESETHANDLER command removes a reset-handler function from the keyboard reset-handler function list. The removed function was previously added to the function list with KBD_ADDRESETHANDLER; it was used to process a Ctrl/left-Amiga/right-Amiga key combination before a hard reset was activated. Once removed, the reset-handler function cannot intercept the reset key combination.

KBD_REMRESETHANDLER is always treated as a queued I/O request and always replies to the task reply-port queue. The results of command execution are found in io_Error; 0 indicates that the command was successful. IOERR_ABORTED indicates that the command was aborted, and IOERR_NOCMD indicates that the io_Command 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 Device and Unit structures that manage Keyboard device unit 0. These can always be copied from the IOStdReq structure initialized by the OpenDevice function call. Set io_Command to KBD-_REMRESTHANDLER and set io_Flags to 0. Initialize io_Data to point to an Interrupt structure representing the keyboard reset-handler function you want to remove from the system. The Interrupt structure was originally designed with the is_Code parameter pointing to the interrupt code for the new function and the is_Data parameter pointing to the data used by that function.

### Discussion

The KBD_ADDRESETHANDLER and KBD_REMRESETHANDLER functions are complements of each other: KBD_ADDRESETHANDLER adds a new keyboard reset-handler function to the system, and KBD_REMRESETHANDLER removes it. Both of these commands work with the Interrupt structure to represent the code and data required to define the function.

## KBD_RESETHANDLERDONE

### Purpose of Command

KBD_RESETHANDLERDONE is used to indicate that a specific reset-handler function has completed its processing. It is dispatched within the function and is usually the last piece of executable code in it. KBD_RESETHANDLERDONE informs the system that the next reset-handler function in the list can begin its processing before a hard reset is finally started.

The IOStdReq structure io_Data parameter is used to point to an Interrupt structure representing the code and data required to define the keyboard reset-handler function that dispatches the KBD_RESETHANDLERDONE command. The results of command execution are found in io_Error; 0 indicates that the command was successful. IOERR_ABORTED indicates that the command was aborted, and IOERR_NOCMD indicates that the io_Command 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 Device and Unit structures that manage Keyboard device unit 0; these can always be copied from the IOStdReq structure initialized by the OpenDevice function call. Set io_Command to KBD_RESETHANDLERDONE and set io_Flags to 0. Initialize io_Data to point to an Interrupt structure representing the keyboard reset-handler function and its data.

### Discussion

KBD_RESETHANDLERDONE terminates execution of a specific reset-handler function. It tells the system that the function has finished performing its reset activities. If the reset-handler function is the last on the reset-handler function list, it terminates the list and the machine's hard-reset sequence will be initiated.

KBD_RESETHANDLERDONE should always be dispatched with the SendIO asynchronous function. The reset-handler function is represented by a software interrupt, and it is illegal to allow a wait operation (such as the one caused by the DoIO function) within a software interrupt.

# 10 The Gameport Device

## Introduction

The Gameport device is responsible for collecting the information an Amiga user inputs from a controller attached to either of the gameport connectors. The Gameport device is ROM-resident; it is loaded automatically into ROM when the WCS ROM is written from the Kickstart disk at boot-up time.

Information coming from the gameport controller usually includes mouse movement input events and events associated with any other type of controller connected to a gameport connector.

The gameport connectors, numbered 1 and 2, are located on the right side of the Amiga. Each is a 9-pin male connector. The front (left) gameport controller, unit 0, is usually dedicated to the Amiga mouse. Mouse button 1, the mouse selection button, is connected to pin 6 of the gameport connector; mouse button 2, the mouse menu button, is connected to pin 9. Table 10.1 shows the pin connections and signals for the gameport connectors. When studying the table, keep in mind that a trackball controller can be declared as a mouse-type controller. The proportional controller is not currently supported by software.

The Gameport device internal routines receive raw gameport-connector signal information and convert it to a set of Gameport device input events. Like all other Amiga input events, each Gameport device input event is represented as an InputEvent structure. The Gameport device internal routines construct one InputEvent structure for each input

**Table 10.1:** *Pin Connections and Signals for Gameport Connectors*

| Pin No. | Absolute Joystick | Mouse Controller | Relative Joystick | Proportional Controllers (2) |
|---|---|---|---|---|
| 1 | Forward | Vertical pulse | Unused | Unused |
| 2 | Back | Horizontal pulse | Unused | Unused |
| 3 | Left | Vertical-quadrature pulse | Button | 1 Left button |
| 4 | Right | Horizontal-quadrature pulse | Button 2 | Right button |
| 5 | Unused | Button 3 (if used) | Potentiometer X | Right potentiometer |
| 6 | Button 1 | Button 1 | Unused | Unused |
| 7 | +5 volts | +5 volts | +5 volts | +5 volts |
| 8 | Ground | Ground | Ground | Ground |
| 9 | Button 2 (if used) | Button 2 | Potentiometer Y | Left potentiometer |

event coming from either active gameport connector. (See Chapter 7 for a discussion of the InputEvent structure.)

The Gameport device places the InputEvent structures into one of two device internal read buffers. The device internal routines maintain a separate buffer for each of the two Gameport device units, which allows input event accumulation. These two buffers are automatically managed by the Gameport device internal routines; they should not be confused with the task-defined buffer used by the GPD_READEVENT command.

Once Gameport device input events are in the internal read buffer, they can be processed in several ways. A task can use the GPD_READEVENT command to place the InputEvent structures into a task-defined buffer, which allows it to process each event as the programmer desires. If the task does not read the InputEvent structures into its own buffer, the Input device receives the input events; they will then be merged with input events coming from the Keyboard and TrackDisk devices and any other sources in the Amiga system. These input events will then be passed on to all of the input-handler functions in order of priority, including Intuition (at priority 50) and the Console device (at priority 0).

The GPD_ASKCTYPE and GPD_SETCTYPE commands allow a task to determine, specify, and change the type of hardware device connected to each gameport connector. These two commands allow the Gameport device internal routines and your tasks to recognize different hardware when it is connected to the gameport. In addition, the GPD_ASKTRIGGER and GPD_SETTRIGGER commands allow each task to determine and specify the trigger conditions necessary for the hardware to generate input event signals. These commands allow the programmer to increase or decrease the threshhold conditions for hardware signals.

The Gameport device has only one device-specific structure: GamePortTrigger. Because it is a simple structure with no pointer parameters or substructures, a diagram of the GamePortTrigger structure is not included in this chapter. However, you should read the discussion of it later in this chapter to see how its parameters are defined.

## Operation of the Gameport Device

Figure 10.1 illustrates the general operation of the Gameport device. The gameport connectors are on the right side of the Amiga as you face it from the front. As you look at the right side, gameport connector 1 is near the front and gameport connector 2 is near the rear. Controller signals that come from the front gameport connector always go to Gameport device unit 0; controller signals that come from the rear gameport connector always go to Gameport device unit 1. These device units can only be opened in exclusive access mode; shared access is not allowed for either unit.

Gameport device unit 0 is usually dedicated to the Input device. All Gameport device events generated by Unit 0 are merged automatically into the entire input event stream by the Input device internal routines; a task cannot use the GPD_READEVENT command to process unit 0 input events. Of course, this mechanism only works if the Input device has been opened previously with an OpenDevice function call somewhere in the task that is using the Gameport device, or if the Gameport device was opened when the Console or Input device was opened.

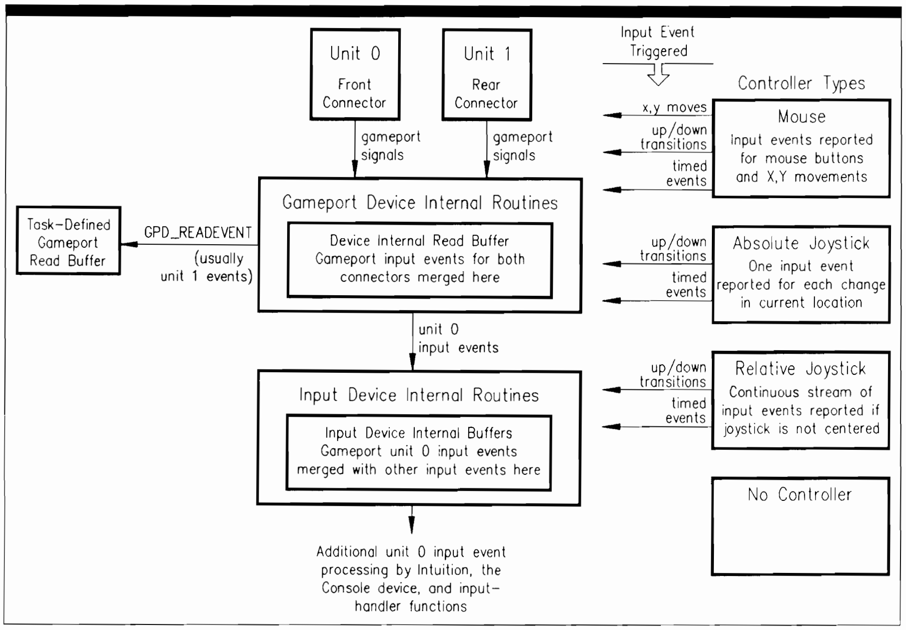

Unit 0
Front Connector
Unit 1
Rear Connector
gameport signals
gameport signals
Input Event Triggered
Controller Types
x,y moves
up/down transitions
timed events
Mouse
Input events reported for mouse buttons and X,Y movements
Gameport Device Internal Routines
Device Internal Read Buffer
Gameport input events for both connectors merged here
Task-Defined Gameport Read Buffer
GPD_READEVENT
(usually unit 1 events)
up/down transitions
timed events
Absolute Joystick
One input event reported for each change in current location
unit 0 input events
Input Device Internal Routines
Input Device Internal Buffers
Gameport unit 0 input events merged with other input events here
up/down transitions
timed events
Relative Joystick
Continuous stream of input events reported if joystick is not centered
No Controller
Additional unit 0 input event processing by Intuition, the Console device, and input-handler functions

**Figure 10.1:** *Operation of the Gameport Device*

Gameport device input events generated by Gameport device unit 1 are not processed by the Input device routines automatically. Instead, they are usually placed in a task-defined buffer with the GPD_READEVENT command. The task can then process these events in any way that the programmer desires; they do not automatically go to the Input device internal routines for further processing.

The Gameport device allows you to use the following types of controllers:

- A mouse (type GPCT_MOUSE). This controller can report input events for one, two, or three mouse buttons and for negative or positive X,Y movements. The input events fall into three classes: X,Y movements represented by a changing set of mouse coordinates; up and down transitions of mouse buttons; and timed input events controlled by each of the mouse buttons.

- An absolute joystick (type GPCT_ABSJOYSTICK). This controller reports one input event for each change in the current location of the joystick. The input events fall into two classes: up and down transitions of the joystick button, and timed input events controlled by each joystick button.

- A relative joystick (type GPCT_RELJOYSTICK). This controller reports a continuous stream of input events if the joystick controller stick is not centered. The input events fall into two classes: up and down transitions of the joystick button, and timed input events controlled by each joystick button.

In addition, the INCLUDE files provide a controller type called GPCT_NO-CONTROLLER. It tells the Gameport device internal routines that no controller is connected to a specific unit, or it tells them to ignore the gameport controller signals coming from the unit.

## Gameport Input Event Processing

All gameport input events are processed by a predefined sequence of routines, as shown in Figure 10.2. The events originate in the gameport hardware as signals, which are passed to the Gameport device internal routines for processing. The general sequence of gameport input event processing is very similar to that of the Keyboard device (see Figure 9.2) except that AmigaDOS does not remove any gameport input events.

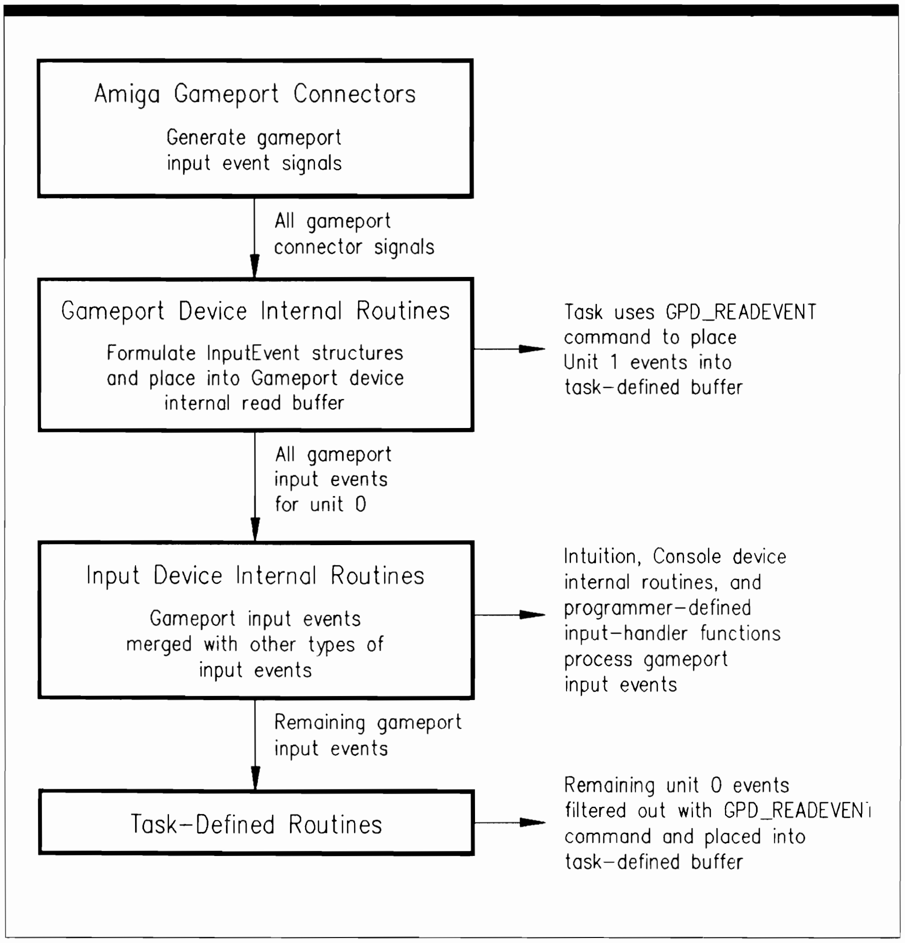

Amiga Gameport Connectors
Generate gameport input event signals
All gameport connector signals
Gameport Device Internal Routines
Formulate InputEvent structures and place into Gameport device internal read buffer
Task uses GPD_READEVENT command to place Unit 1 events into task-defined buffer
All gameport input events for unit 0
Input Device Internal Routines
Gameport input events merged with other types of input events
Intuition, Console device internal routines, and programmer-defined input-handler functions process gameport input events
Remaining gameport input events
Task-Defined Routines
Remaining unit 0 events filtered out with GPD_READEVENT command and placed into task-defined buffer

**Figure 10.2:** *Input Event Processing for the Gameport Device*

## Gameport Device Commands

You can program the Gameport device with five device-specific commands and one standard device command. All six commands support both QuickIO and queued I/O; most also support immediate-mode operation. All commands affect the IOStdReq structure io_Error parameter, and CMD_CLEAR also affects the contents of the Gameport device internal read buffer.

### Sending Commands to the Gameport Device

Figure 10.3 depicts the general scheme used to send commands to the Gameport device routines. The lines with arrows represent the parameters you should initialize and those returned by the device internal routines.

The Gameport device programming process consists of three phases:

1. Structure preparation. You have complete control over this phase; here, you initialize parameters in the IOStdReq structure in preparation for sending a command to the Gameport device internal routines. These parameters include the set of parameters required by most devices. The choice of parameters to initialize depends on the specific command you plan to send to a Gameport device unit. Taken together, these parameters provide an information path to the data needed by the Gameport device routines to process the command.

2. Gameport device internal processing. The only part you play in this phase is to send the command to the device using the BeginIO, DoIO, or SendIO function. Control then passes to the device and system internal routines.

3. Command output parameter processing. The system and Gameport device internal routines have complete control over this phase. The results of Gameport device command processing have been returned to the task that originally dispatched the command. If the I/O request was unsuccessful as QuickIO, it was processed when it moved to the top of the Gameport device-unit request queue; it is now in the task reply-port queue

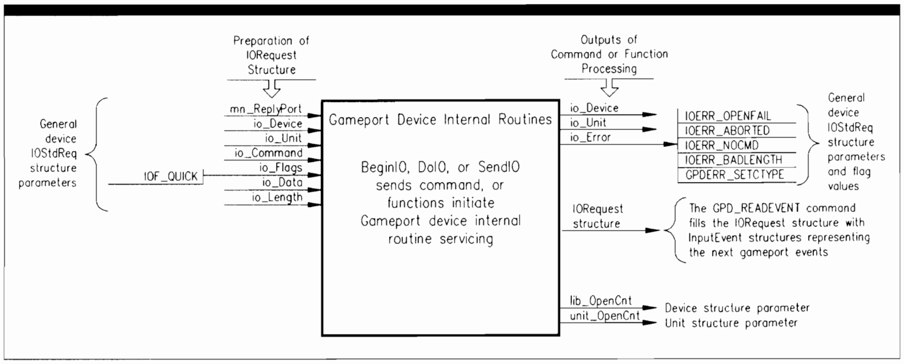

Preparation of IORequest Structure
Outputs of Command or Function Processing
General device IOStdReq structure parameters
IOF_QUICK
mn_ReplyPort
io_Device
io_Unit
io_Command
io_Flags
io_Data
io_Length
Gameport Device Internal Routines
BeginIO, DoIO, or SendIO sends command, or functions initiate Gameport device internal routine servicing
io_Device
io_Unit
io_Error
IOERR_OPENFAIL
IOERR_ABORTED
IOERR_NOCMD
IOERR_BADLENGTH
GPDERR_SETCTYPE
General device IOStdReq structure parameters and flag values
IORequest structure
The GPD_READEVENT command fills the IORequest structure with InputEvent structures representing the next gameport events
lib_OpenCnt
unit_OpenCnt
Device structure parameter
Unit structure parameter

**Figure 10.3:** *Gameport Device Command and Function Processing*

awaiting the task's processing. If the request was successful as QuickIO, it was not queued but came back directly to the requesting task. The parameters still direct you to appropriate data for your task.

As the figure shows, several Gameport device commands provide output parameters; in addition, the GPD_READEVENT command fills the IOStdReq structure io_Data buffer with InputEvent structures for the next series of gameport input events.

Figure 10.3 also depicts the parameters that play a part in Gameport device function setup and processing. The OpenDevice and CloseDevice functions both affect the Unit structure unit_OpenCnt parameter and the Device structure lib_OpenCnt parameter; OpenDevice also affects the io_Error parameter.

## Structures for the Gameport Device

The Gameport device software system uses only one structure: GamePortTrigger. It allows the GPD_ASKTRIGGER and GPD_SETTRIGGER commands to determine and initialize the trigger conditions that will cause gameport connector signals to generate input events.

### The GamePortTrigger Structure

The GamePortTrigger structure is defined as follows:

```
struct GamePortTrigger {
   UWORD gpt_Keys;
   UWORD gpt_Timeout;
   UWORD gpt_XDelta;
   UWORD gpt_YDelta;
};
```

The parameters in the GamePortTrigger structure have the following meanings:

- gpt_Keys. Your task should initialize this parameter to GPTF_DOWNKEYS if you want a controller's button-down transitions to trigger a Gameport device input event. Initialize this to GPTF_UPKEYS in order for a controller's button-up transitions to trigger a Gameport device input event.

- gpt_Timeout. This parameter represents the time interval that will trigger a Gameport device input event if exceeded. The time interval is measured in vertical-blanking intervals; each interval is 1/60 of a second.

- gpt_XDelta. This parameter sets a controller's X-direction movement, measured in horizontal screen-resolution pixels. If exceeded, it triggers a Gameport device input event.

- gpt_YDelta. This parameter sets a controller's Y-direction movement, measured in vertical screen-resolution pixels. If exceeded, it triggers a Gameport device input event.

## USE OF FUNCTIONS

## *CloseDevice*

### Syntax of Function Call

**CloseDevice (iOStdReq)**
**A1**

### Purpose of Function

This function closes access to unit 0 or 1 of the Gameport device. If this is the last Close-Device function call for the units in the task and the Console and Input devices have also been closed in the task, the Gameport device will be closed.

CloseDevice sets the IOStdReq structure io_Device and io_Unit parameters to −1; a task cannot use the structure again until these parameters are reinitialized by OpenDevice. CloseDevice also reduces the Device structure lib_OpenCnt and Unit structure unit_OpenCnt parameters by 1 to indicate that one less task is using the Gameport device unit.

### Inputs to Function

| | |
|---|---|
| **iOStdReq** | A pointer to an IOStdReq structure |

### Discussion

CloseDevice terminates access to a set of Gameport device routines for a specific unit of the device. A task should always verify that all of its I/O requests have been replied by the Gameport device internal routines before it calls CloseDevice. It can do so by using the GetMsg, Remove, CheckIO, and WaitIO functions to see what I/O requests are currently in the task reply-port queue.

When a task is done with its Gameport device operations, it should close the device with a call to the CloseDevice function. This frees RAM that might be needed by the system. Another task that wants to use the Gameport device can then open, use, and close it, and the sequence can be repeated again and again.

Always remember that the Gameport device is opened indirectly when the Input or Console device is opened; it is opened automatically by AmigaDOS as well. The procedure you use to open the Gameport device should determine how you proceed with your Gameport device CloseDevice calls.

## OpenDevice

### Syntax of Function Call

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

### Purpose of Function

The OpenDevice function opens access to the internal routines of Gameport device unit 0 or 1. The Gameport device can also be opened automatically either by AmigaDOS or by the Console and Input devices when they are opened directly or indirectly.

OpenDevice automatically initializes a Unit structure to manage Gameport device unit 0 or 1. The Unit structure contains a MsgPort substructure representing the device-unit request queue for all commands dispatched to that unit. Because each unit of the Gameport device operates in exclusive access mode, tasks send their nonQuickIO requests to this message port; the requests are all queued in the same queue.

Once OpenDevice has successfully opened the Gameport device, it initializes certain Gameport device internal parameters to their most recently specified or default values. OpenDevice also increments the Device structure lib_OpenCnt parameter and the Unit structure unit_OpenCnt parameter. OpenDevice requires a properly initialized reply port with a task signal bit allocated to that port if the calling task wants to be signaled when the Gameport device routines reply to the task reply-port queue. The results of command execution are found in the following parameters:

- io_Device. This points to a Device structure that will manage unit 0 or 1 once it has been opened. The Device structure contains all the information necessary to manage the Gameport device unit and to reach the data and routines in the Gameport device library.

- io_Unit. This points to a Unit structure that will be used to define and manage a MsgPort structure for Gameport device unit 0 or 1. The MsgPort structure represents the device-unit request queue.

- io_Error. 0 indicates that the requested open succeeded. IOERR_OPENFAIL indicates that the Gameport device could not be opened; this usually means that there was not enough memory to accommodate the Device and Unit structures and the Library routines. IOERR_NOCMD indicates that io_Command was specified incorrectly.

## Inputs to Function

| | |
|---|---|
| **"gameport.device"** | A pointer to a null-terminated string representing the name of the Gameport device |
| **unitNumber** | Either 0 for the front (unit 0) gameport controller or 1 for the rear (unit 1) gameport controller |
| **iOStdReq** | A pointer to an IOStdReq structure |
| **0** | Indicates that the flags argument is not used by the Gameport device |

## Preparation of the IOStdReq Structure

Initialize mn_ReplyPort to point to a MsgPort structure representing the task reply port that will receive the I/O request reply from the Gameport device when it has finished processing any of the Gameport device commands. Initialize all other IOStdReq substructure parameters to 0, or copy them from the IOStdReq structure of a previous OpenDevice call. Initialize io_Command to 0, or set it to GPD_READEVENT if the task should open the Gameport device and dispatch a GPD_READEVENT command immediately.

If the CreateStdIO function is used to create the IOStdReq structure (see Chapter 2), it will automatically return a pointer to an IOStdReq structure. In this case, no typecasting will be necessary for the Gameport device.

## Discussion

The OpenDevice function opens the Gameport device internal routines for access by a task. Gameport device units can only be opened in exclusive access mode, the default; shared access is not allowed. Keep in mind that the Gameport device is opened automatically by AmigaDOS or by the Console and Input device when a system-generated OpenDevice call for these two devices is executed.

The IOStdReq structure parameters are set to their default values the first time the Gameport device is opened in a task. This means that most of them are initialized to 0. However, the mn_ReplyPort parameter should always be initialized before OpenDevice is called so that it points to a MsgPort structure representing the task reply-port queue.

Once the Gameport device is opened, other parameters in the IOStdReq structure can be initialized to define I/O requests. Any parameters that are not explicitly initialized will retain their previous values or be initialized to their default values. If a calling task wants to use values other than the defaults for these parameters, it should initialize them after the OpenDevice function call returns.

OpenDevice can be called to open the Gameport device and to dispatch a series of GPD_READEVENT and other commands to the Gameport device routines. The commands then gather information from the Amiga gameport controllers and send it to the task, the Input device, or the Console device for further processing. Once a task has finished its Gameport device processing, it can close the device using a call to CloseDevice.

## STANDARD DEVICE COMMANDS

## CMD_CLEAR

### Purpose of Command

CMD_CLEAR clears the Gameport device internal read buffer for a specified unit. This buffer is only used to store Gameport device input events (in the form of InputEvent structures) generated by gameport controller signals. Once the read buffer is cleared, subsequent GPD_READEVENT commands can proceed from a known empty buffer condition. CMD_CLEAR allows QuickIO and only replies if the request was queued or if QuickIO was not successful. IOERR_NOCMD indicates that io_Command 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 Device and Unit structures that manage Gameport device unit 0 or unit 1; these parameters can be copied from the IOStdReq structure initialized by the OpenDevice function call. Also initialize io_Command to CMD_CLEAR. Set io_Flags to IOF_QUICK for QuickIO; otherwise, set it to 0. QuickIO may or may not succeed, depending on conditions in the system when the command was dispatched.

### Discussion

The Gameport device automatically maintains one internal device buffer, which it uses to store Gameport device input events. The CMD_CLEAR command clears the current contents of this buffer. If a task is executing a series of GPD_READEVENT commands to place input events into a task-defined buffer and wants to be sure the Gameport device internal read buffer is in a known state before proceeding, it should first dispatch CMD_CLEAR to zero all bytes in that buffer. Then subsequent GPD_READEVENT commands will not encounter characters left over from earlier input events. The Gameport device does not have a CMD_WRITE command and therefore uses no internal write buffers.

## DEVICE-SPECIFIC COMMANDS

## *GPD_ASKCTYPE*

### Purpose of Command

The GPD_ASKCTYPE command determines the controller type currently connected to Gameport device unit 0 or unit 1. The Gameport device internal routines then provide the address of this information to the task in the IOStdReq structure io_Data parameter. The Gameport device internal routines and the task routines are then consistent, and the task can properly interpret the gameport connector signals.

GPD_ASKCTYPE is always an immediate-mode command. It replies to the task reply-port queue if a requested QuickIO was unsuccessful. The results of command execution are found in io_Error, where 0 indicates that the task was successful. IOERR-_ABORTED indicates that the command 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 Device and Unit structures that manage Gameport device unit 0 or unit 1; these parameters can be copied from the IOStdReq structure initialized by the OpenDevice function call. Set io_Command to GPD_ASKCTYPE. Also initialize the following command-specific parameters:

- io_Flags. Set this to IOF_QUICK for QuickIO; otherwise, set it to 0. QuickIO may or may not succeed, depending on conditions in the system when the command was dispatched.
- io_Length. Initialize this to 1, indicating that the value returned in the location addressed by the io_Data parameter is one byte in length.
- io_Data. Initialize this to point to a byte variable in which the single-byte result returned by GPD_ASKCTYPE will be placed. When GPD_ASKCTYPE completes execution, this byte variable will have one of five values, as discussed in the next section.

### Discussion

The GPD_ASKCTYPE and GPD_SETCTYPE commands allow a task to determine and to change the controller type connected to a Gameport device unit. GPD_ASKCTYPE tells the task what controller types are connected, and GPD_SETCTYPE changes those type specifications when the specific hardware connected to a Gameport device unit has been changed. A task and the Gameport device internal device routines must always agree on the controller type in order for the gameport controller signals to be properly and consistently interpreted.

At present, the Amiga supports controller types GPCT_NOCONTROLLER, GPCT_MOUSE, GPCT_RELJOYSTICK, and GPCT_ABSJOYSTICK. Each of these types is identified by a specific constant (0, 1, 2, and 3, respectively) in the Gameport.h INCLUDE file. In addition, GPCT_ALLOCATED, with an assigned value of −1, indicates that a task has asked for a controller type that has already been allocated to another task. It is required because the Gameport device units are used in exclusive access mode only. See the earlier discussion in this chapter for more on these controller types.

## *GPD_ASKTRIGGER*

### Purpose of Command

The GPD_ASKTRIGGER command allows a task to determine the current trigger conditions that must be met by a Gameport device controller before a pending Gameport input event will occur. GPD_ASKTRIGGER is an immediate-mode command and always replies to the task reply-port queue if it was queued or if QuickIO was unsuccessful. The results of command execution are found in io_Error, where 0 indicates that the task was successful. IOERR_ABORTED indicates that the command 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 Device and Unit structures that manage Gameport device unit 0 or unit 1; these parameters can be copied from the IOStdReq structure initialized by the OpenDevice function call. Set io_Command to GPD_ASKTRIGGER. Also initialize the following command-specific parameters:

- io_Flags. Initialize this to IOF_QUICK for QuickIO; otherwise, set io_Flags to 0. QuickIO may or may not succeed, depending on conditions in the system when the command was dispatched.

- io_Length. Initialize this to the size (in bytes) of the GamePortTrigger structure. A task can use the C language sizeof operator to initialize this parameter.

- io_Data. Initialize this to point to the GamePortTrigger structure representing the current trigger conditions for the gameport controller unit. The task can read the structure parameters to determine the trigger conditions.

### Discussion

The GPD_ASKTRIGGER and GPD_SETTRIGGER commands allow a task to determine and to change the current gameport trigger conditions for a hardware device connected to a Gameport device unit. GPD_ASKTRIGGER determines the trigger conditions, and the device internal routines use the information to determine if a new Gameport device input event has occurred. If the trigger conditions are satisfied, the Gameport device internal routines will formulate a new InputEvent structure to represent the event; it will then be queued in the Gameport device internal read buffer. If the trigger conditions are not satisfied, a new InputEvent structure will not be formulated.

The GPD_ASKTRIGGER command does not relate directly to the GPD_READEVENT command. However, if trigger conditions are such that the Gameport device internal read buffer contains one or more input events because the current trigger conditions have been satisfied one or more times, a task can execute GPD_READEVENT to read them into its own task-defined buffer.

## *GPD_READEVENT*

### Purpose of Command

The GPD_READEVENT command allows a task to read the next series of Gameport device input events from the Gameport device internal read buffer into a task-defined buffer. The task can then process those events as the programmer desires. GPD_READEVENT uses the IOStdReq structure io_Length and io_Data parameters to specify the task-defined buffer; its size is determined by the number of InputEvent structures to be placed in it.

If no Gameport device input events are currently queued in the Gameport device internal read buffer, GPD_READEVENT will not be satisfied and will not read any input events into the task-defined buffer. In this case, the IOStdReq structure io_Error parameter will indicate that an error has occurred. However, if there are some Gameport device input events in the internal read buffer that have not yet been read, but not as many as indicated by the io_Length parameter, GPD_READEVENT will be satisfied with those currently available, and the io_Error parameter will be 0 when the GPD_READEVENT request is replied.

GPD_READEVENT allows QuickIO and always replies to the task reply-port queue if it was queued or if QuickIO was unsuccessful. The results of command execution are found in io_Error, where IOERR_ABORTED indicates that the command 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 Device and Unit structures that manage Gameport device unit 0 or unit 1; these parameters can be copied from the IOStdReq structure initialized by the OpenDevice function call. Set io_Command to GPD_READEVENT. Also initialize the following command-specific parameters:

- io_Flags. Initialize this to IOF_QUICK for QuickIO; otherwise, set it to 0. QuickIO may or may not succeed, depending on conditions in the system when the command was dispatched.
- io_Length. Initialize this to the size (in bytes) of the task-defined buffer. To determine its size, multiply the number of InputEvent structures by the size of the InputEvent structure. A task can use the C language sizeof operator to initialize this parameter.
- io_Data. Initialize this to point to the task-defined buffer. When GPD_READEVENT returns, the buffer will contain a series of InputEvent structures representing the new gameport input events that have been copied from the Gameport device internal read buffer.

## Discussion

GPD_READEVENT places InputEvent structures for one or more Gameport device input events into a task-defined buffer. The task that dispatches the GPD_READEVENT command can then process those input events as the programmer desires. Each InputEvent structure in the task-defined buffer will contain the following parameters:

- ie_NextEvent. This points to the next InputEvent structure in the linked list; it will be 0 if the InputEvent structure represents the last structure in the list.
- ie_Class. This is the class of the gameport event; it is IECLASS_RAWMOUSE for raw mouse movement events.
- ie_SubClass. This is 0 if the Gameport device input event came from the front (unit 0) gameport controller; it is 1 if the input event came from the rear (unit 1) gameport controller.

- ie_Code. This contains the next controller-button up or down report. It is either IECODE_LBUTTON, IECODE_RBUTTON, IECODE_MBUTTON, or IECODE_NBUTTON. See the Inputevent.h INCLUDE file for more information. A hexadecimal 0xFF value indicates no report.
- ie_Qualifier. This is the qualifier of the gameport input event. It is either IEQUALIFIER_LBUTTON, IEQUALIFIER_RBUTTON, IEQUALIFIER_MBUTTON, or IEQUALIFIER_RELATIVEMOUSE. Again, see the Inputevent.h INCLUDE file for more information.
- ie_X. This is the current controller X position when the Gameport device input event occurred. It is measured in the horizontal resolution of the display screen.
- ie_Y. This is the current controller Y position when the Gameport device input event occurred. It is measured in the vertical resolution of the display screen.
- ie_TimeStamp. This is the time since the last Gameport device input event occurred. It is measured in the number of video frames since the last input event and is a multiple of 1/60 of a second. The Gameport device internal routines place the number of frames in the TimeVal structure tv_Secs parameter. Recall that the TimeVal structure usually contains the tv_Secs (seconds) and tv_Micros (microseconds) values. Here, however, the tv_Micros value has no meaning; instead, the Gameport device initializes the tv_Secs parameter to the number of frames since the last Gameport device input event.

Gameport input events are usually passed automatically to the Input device. However, with the GPD_READEVENT command, your tasks can remove events from the Gameport device internal read buffer, place them in their own buffers, and process them as you wish.

Gameport device routines will normally queue more than one input event into the Gameport device internal read buffer. These events will continue to queue in the buffer until a task reads them with GPD_READEVENT or the Input device reads them automatically and passes them to Intuition and the Console device. If neither the task nor the Input device reads the events, the Gameport device internal read buffer will overflow; any Gameport device input events beyond those already stored in the buffer will be lost.

## GPD_SETCTYPE

### Purpose of Command

The GPD_SETCTYPE command allows a task to change the controller type connected to Gameport device unit 0 or 1. This change makes the Gameport device internal software system consistent with the hardware system and allows a task to inform the Gameport device internal routines of the current hardware status. A task and the Gameport device internal routines can then interpret signals properly when the controller has been changed. The controller type that the task wants to change is placed into the IOStdReq structure io_Data parameter before the GPD_SETCTYPE command is dispatched. After the GPD_SETCTYPE command executes, the Gameport device internal routines are informed immediately of the new controller type.

GPD_SETCTYPE is always an immediate-mode command and always replies to the task reply-port queue if it was queued or if QuickIO was not successful. The results of command execution are found in io_Error, where 0 indicates that the task was successful. IOERR_ABORTED indicates that the command 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. GPDERR_SETCTYPE indicates that the specified controller is not valid at this time.

## 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 Unit structures that manage Gameport device unit 0 or unit 1; these parameters can be copied from the IOStdReq structure initialized by the OpenDevice function call. Set io_Command to GPD_SETCTYPE. Also initialize the following command-specific parameters:

- io_Flags. Initialize this to IOF_QUICK for QuickIO; otherwise, io_Flags should be set to 0. QuickIO may or may not succeed, depending on conditions in the system when the command was dispatched.
- io_Length. Initialize this to 1, indicating that the value placed into the IOStdReq structure io_Data parameter is one byte in length.
- io_Data. Initialize this to point to the byte variable in which the specified new controller type is stored. The byte variable will have one of five values: −1, 0, 1, 2, or 3. See the GPD_ASKCTYPE command discussion for the meanings of these values.

## Discussion

The type of controller hardware connected to unit 0 and unit 1 must be known by both the Gameport device internal routines and the task routines, and they must always agree on the controller type connected. This allows the gameport controller signals to be properly and consistently interpreted by both sets of routines. GPD_SETCTYPE allows a task to specify the controller type connected to a specific Gameport device unit. GPD_ASKCTYPE tells the task what controller type is connected to each unit, and GPD_SETCTYPE changes the controller type in software when the hardware controller connected to the Gameport device unit has changed.

## GPD_SETTRIGGER

### Purpose of Command

The GPD_SETTRIGGER command allows a task to change the current trigger conditions that must be met by a gameport controller before a Gameport device input event will occur. GPD_SETTRIGGER is an immediate-mode command and always replies to the task reply-port queue if it was queued or if QuickIO was not successful. The results of command execution are found in io_Error, where 0 indicates that the task was successful. IOERR_ABORTED indicates that the command 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 Device and Unit structures that manage Gameport device unit 0 or unit 1; these parameters can be copied from the IOStdReq structure initialized by the OpenDevice function call. Set io_Command to GPD_SETTRIGGER. Also initialize the following command-specific parameters:

- io_Flags. Initialize this to IOF_QUICK for QuickIO; otherwise, set it to 0. QuickIO may or may not succeed, depending on conditions in the system when the command was dispatched.
- io_Length. Initialize this to the size (in bytes) of the GamePortTrigger structure. A task can use the C language sizeof operator to initialize this parameter.
- io_Data. Initialize this to point to the GamePortTrigger structure representing the new trigger conditions for the gameport controller unit.

### Discussion

The GPD_ASKTRIGGER and GPD_SETTRIGGER commands allow a task to determine and change (set) the current gameport trigger conditions for a hardware device connected to a Gameport device unit. GPD_SETTRIGGER changes the gameport-controller trigger conditions; the Gameport device internal routines use this information to determine if a new Gameport device input event has occurred.

If the new trigger conditions are satisfied, the Gameport device internal routines will create a new InputEvent structure, which will be queued in the Gameport device internal read buffer. If the new trigger conditions are not satisfied, a new InputEvent structure will not be formulated.

11

# The Printer Device

## Introduction

The Printer device is responsible for sending printable characters and printer control codes to a printer connected to the serial or parallel port. At the time of this writing, the Amiga supports the following printers:

Alphacom Alphapro
Apple ImageWriter II
Canon PJ 1080a
CBM MPS1000
Diablo 630
Diablo C 150
Diablo Advantage D25
Epson
Epson JX-80
101 Epson RX-80
HP LaserJet
HP LaserJet Plus
Juki 5510
Okidata 20
Okidata Microline 92
Okidata Microline 192
Okidata Microline 292
Qume LetterPro 20

Amiga also supports a generic printer driver, which allows you to use many commercially available printers not shown on this list. Drivers are found in the Workbench disk DEVS: directory; if the driver is not there, it is available from another source. The Apple ImageWriter II, the Epson printers, the HP LaserJet Plus, and the Okidata printers are dot-matrix printers. The Apple ImageWriter II, Canon PJ 1080a, Diablo C 150, Epson JX-80, Juki 5510, Okidata 20, and Okidata Microline 292 can produce color printouts; the Okidata 20 can produce color text.

The Printer device is disk-resident; it must be loaded into RAM from disk. Its internal routines can be accessed directly by a task, or in AmigaDOS through the SER: file (for a printer connected to the serial port), the PAR: file (for a printer connected to the parallel port), or the PRT: file (for a printer connected to either the serial or parallel port). Specific printer selections and options are made from the Workbench Preferences screen, and the three AmigaDOS files (SER:, PAR:, and PRT:) can be called from an AmigaDOS CLI.

## Operation of the Printer Device

The Printer device has only one unit. It queues command requests for processing by the Printer device internal routines. Figure 11.1 illustrates the general operation of this device.

There are four ways to access the Printer device internal routines. Three ways provide an indirect route, and one provides a direct route. In each case, a continuous stream of control-code commands and printable characters can be sent to the Printer device internal routines. (Control-code commands consist of instructions to a specific printer—to reset the printer hardware, initialize the printer with default settings, send a linefeed character, start italic printing, and so on.) The first two of the following methods are *not* recommended:

1. Access the SER: file from an AmigaDOS CLI (command-line interface). With this method you can also access the Printer device from the Workbench Preferences screen. However, you must access the Printer device internal routines from within a

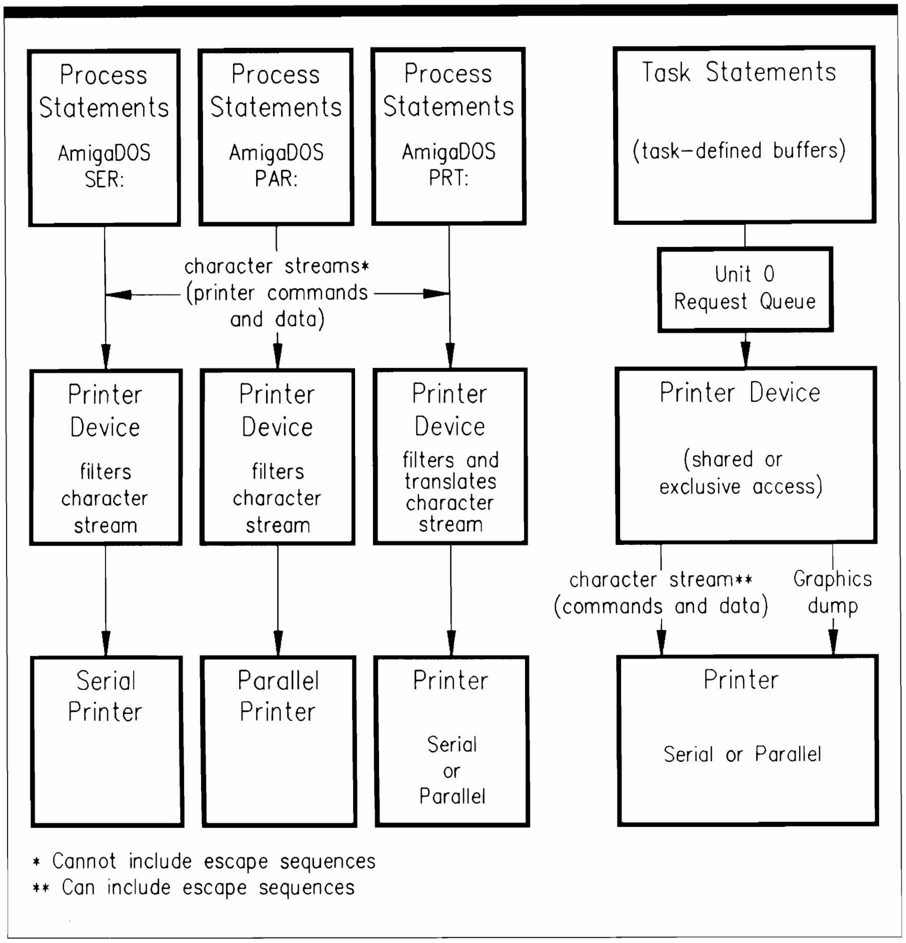

Process Statements AmigaDOS SER:
Process Statements AmigaDOS PAR:
Process Statements AmigaDOS PRT:
Task Statements (task-defined buffers)
character streams* (printer commands and data)
Unit 0 Request Queue
Printer Device filters character stream
Printer Device filters character stream
Printer Device filters and translates character stream
Printer Device (shared or exclusive access)
character stream** (commands and data)
Graphics dump
Serial Printer
Parallel Printer
Printer Serial or Parallel
Printer Serial or Parallel
* Cannot include escape sequences
** Can include escape sequences

**Figure 11.1:** *Operation of the Printer Device*

process; it is not possible to access them from within a task. A continuous character stream will be sent from AmigaDOS to the Printer device internal routines, which will filter the stream and eliminate certain characters. The character stream will not be translated, and you will not be able to send a raster bitmap display to the printer.

2. Access the PAR: file from an AmigaDOS CLI. This method also allows you to access the Printer device from the Workbench Preferences screen. However, you must access the Printer device internal routines from within a process. The internal routines will filter the character stream, eliminating certain characters, but they will not translate the character stream. You will not be able to send a raster bitmap display to the printer.

3. Access the PRT: file from an AmigaDOS CLI. Once again, with this method you can also access the Printer device from the Workbench Preferences screen, but you must access its internal routines from within a process. The internal routines will filter the character stream, eliminating certain characters. With this method, characters in the stream will also be translated into another set of characters. However, you will not be able to send a raster bitmap display to the printer.

4. Access the Printer device internal routines directly from a task. With this method, a continuous stream of characters is sent to the Printer device internal routines from a task-controlled buffer. The task is defined and controlled by a Task structure; there is no need to deal with processes. If dispatched with the PRD_PRTCOMMAND and CMD_WRITE commands, the control-code characters will be translated into specific control characters for the printer selected on the Workbench Preferences screen. A task can also use the PRD_RAWWRITE command to send characters that are not translated by the Printer device internal routines.

With all four methods, the Printer device internal routines can only be used in exclusive access mode. Since this device communicates with external hardware, multiple tasks must not use the hardware simultaneously.

## Sending Control Codes to a Printer

The Amiga can send any standard 7-bit ANSI 3.64 character to a connected printer. When preceded by an escape character, each character or group of characters can represent a printer control code, which is an escape sequence that a printer understands. The Amiga can work with 75 different printer control codes. These include codes defined by the International Standards Organization and the Digital Equipment Corporation, as well as codes designed specifically for the Amiga, including the initialize, subscript-on, and subscript-off codes described in the *ROM Kernel Manual.*

Every printer has its own set of control-code characters. Therefore, it is the responsibility of each printer driver in the DEVS: directory to translate nonstandard escape sequences into control codes. The translation capability must be built into any printer driver you use. Once a specific driver is created and placed in the DEVS: directory printer file, it can be controlled from an AmigaDOS CLI, which requires a process, or from within a task, which does not require a process.

There are two procedures for controlling the printer directly from an AmigaDOS CLI. The first procedure is as follows:

1. Select the appropriate printer from the Workbench Preferences screen.

2. Redirect the keyboard input to the printer by typing

   ```
   Copy * to PRT:
   ```

   and pressing Return. This opens a keyboard-generated file that will contain the characters you type. Then wait until the file is open and the disk access stops.

3. At the CLI prompt, type an escape-character sequence. For example, to turn on italic printing, press the Escape key and type

   ```
   [3m
   ```

   This information will not be echoed to the screen but instead will be placed in the keyboard-generated file.

4. Hold down the Ctrl key and press the Backslash key to terminate the keyboard-generated file. If your printer supports italic printing and the printer driver was properly designed and programmed, any information you now send to the printer will be printed in italic. If you want to print a file named MyFile with italic characters, for example, type

   ```
   Copy Myfile to PRT:
   ```

   and press the Return key.

The second procedure for controlling the printer from AmigaDOS is similar, except that it allows you to automate the process. You create a number of small control-code files (called printer command files) that you send to the printer when you need to reconfigure it for new printing options. You can place these files on a disk and read them into a RAM disk as part of the startup-sequence file. For example, you could design a number of printer command files to do the following:

- Create a file named ION to turn italic printing on.
- Create a file named IOFF to turn italic printing off.
- Create a file named CON to turn condensed printing on.
- Create a file named COFF to turn condensed printing off.

Each of these files would have a set of control-code sequences. Then, when you wanted to turn the features on or off, you would type the appropriate command, such as

```
Copy ram:ION to PRT:
```

to turn italic printing on, or

```
Copy ram:IOFF to PRT:
```

to turn italic printing off.

To control a printer from within a task, you select the printer from the Workbench Preferences screen and send it printer control codes by using PRD_PRTCOMMAND (for a single printer control code) or CMD_WRITE (for a series of printer control codes). Any control code you send will be translated by the correct printer driver. You could also send printer control codes with the PRD_RAWWRITE command if you did not want them to be translated.

## Printer Device Commands

The Printer device has three device-specific and five standard device commands. All but two of these commands support QuickIO; PRD_PRTCOMMAND and CMD_WRITE do not. Three commands (CMD_FLUSH, CMD_START, and CMD_STOP) also support immediate-mode operation. All commands affect the PrinterIO union IOStdReq, IODRPReq, or IOPrtCmdReq substructure io_Error parameter.

### Sending Commands to the Printer Device

Figures 11.2(a) and (b) depict the general scheme used to dispatch commands to the Printer device. In Figure 11.2(a), the lines with arrows represent the parameters you should initialize; in Figure 11.2(b), they represent the parameters returned by the Printer device internal routines.

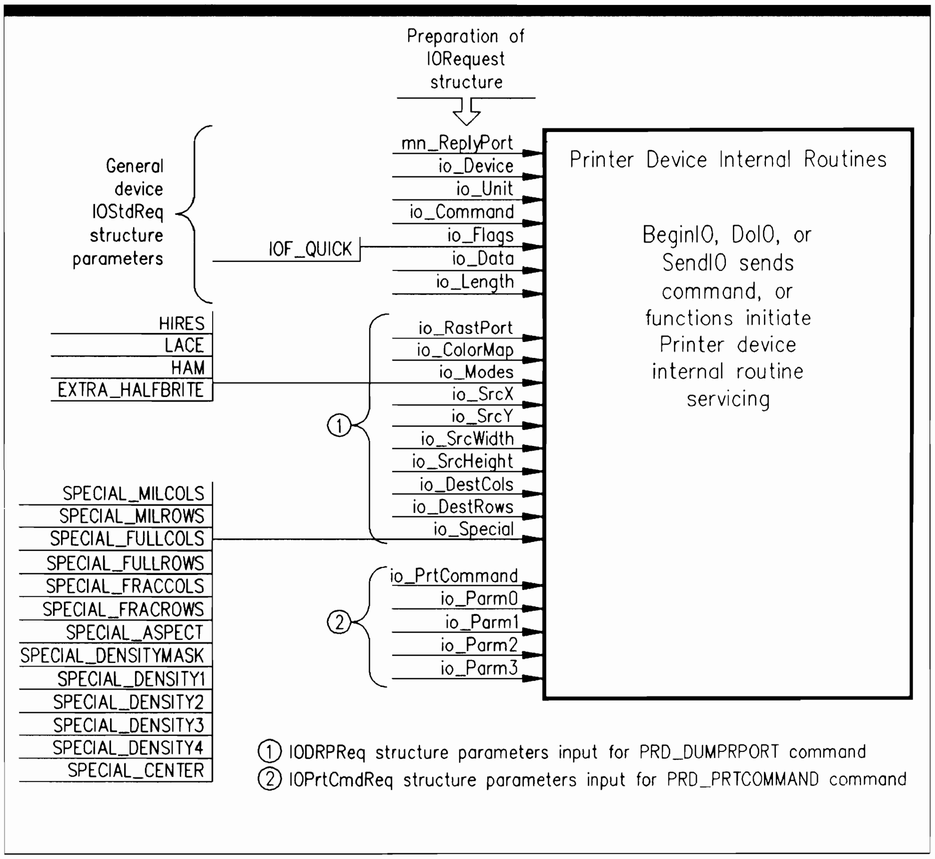


**Figure 11.2(a):** *Printer Device Command and Function Processing (Specifications)*

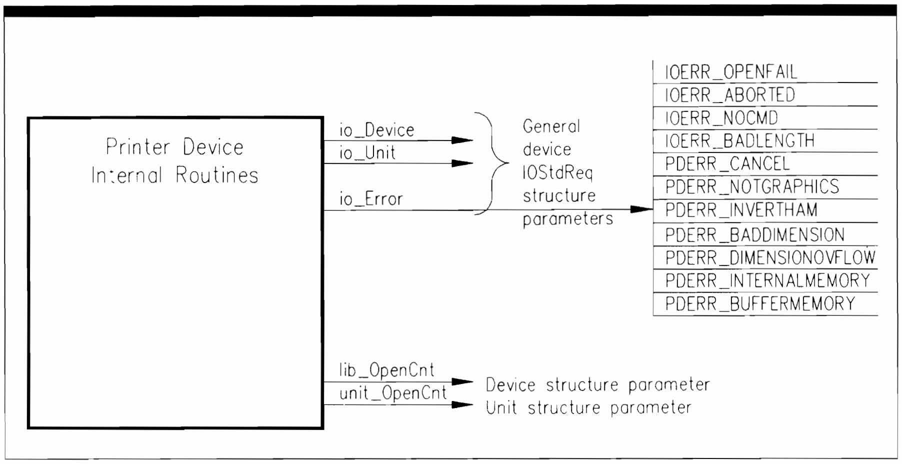


**Figure 11.2(b):** *Printer Device Command and Function Processing (Outputs)*

The Printer device programming process consists of three phases:

1. Structure preparation. The programmer has complete control over this phase. Here, you initialize parameters in the IOStdReq, IOPrtCmdReq, or IODRPReq substructure in preparation for dispatching a command. These parameters include those required by most devices, as well as parameters for the IODRPReq and IOPrtCmdReq substructures. The choice of parameters depends on the specific command you plan to dispatch to the Printer device. These parameters provide an information path to the data needed by the Printer device internal routines to process a command.

2. Printer device internal routine processing. The only part you play in this phase is to dispatch the command to the device using the BeginIO, DoIO, or SendIO function. Once the function begins executing, control passes to the device and system internal routines.

3. Command output parameter processing. The system and Printer device internal routines have complete control over the values found in this phase. Here, the results of command processing have been returned to the task that issued the command. If the I/O request was not a QuickIO or immediate-mode request, it was processed when it moved to the top of the device-unit request queue and is now in the task reply-port queue awaiting the task's processing. If the request was a successful QuickIO request, it was not queued but came directly back to the requesting task. The parameters still direct you to appropriate data for your task.

All commands provide output parameters. These are returned by the PrinterIO union request substructure io_Error parameter.

Figures 11.2(a) and (b) also depict the parameters that play a part in Printer device function setup and processing. OpenDevice and CloseDevice both affect the Unit structure unit_OpenCnt parameter and the Device structure lib_OpenCnt parameter; OpenDevice also affects the io_Error parameter.

The procedures used to dispatch commands to the Printer device internal routines are different from those for most Amiga devices. The Exec functions (BeginIO, DoIO, and SendIO) must work with a special C language union named PrinterIO. This union has the particular construction that the Printer device internal routines were programmed to expect when a command is dispatched.

## The PrinterIO Union

When a task wants to dispatch a Printer device command, it must initialize a PrinterIO union. This is usually done by using the Exec-support library CreateExtIO function before OpenDevice is called. The same PrinterIO union can be used for any number of Printer device commands; the procedure does not differ from the formulation procedures for other structures.

Recall that a C language union is similar to a C language structure, except that each of its component parts (in this case, substructures) are interpreted to occupy the same RAM locations. The IOStdReq, IODRPReq, and IOPrtCmdReq substructures all occupy the initial bytes of the same RAM area. This arrangement is understood by the C language compiler; it properly compiles and links when it sees this construction, which allows the system to save RAM.

The PrinterIO union is defined as follows:

```
union {
   struct IOStdReq ios;
   struct IODRPReq iodrp;
   struct IOPrtCmdReq iopc;
} PrinterIO;
```

The IOStdReq substructure represents the I/O request structure for the standard device commands and the PRD_RAWWRITE command. The IODRPReq substructure represents the PRD_DUMPRPORT command, and the IOPrtCmdReq substructure represents the PRD_PRTCOMMAND command. A task must initialize the appropriate parameters in these substructures to dispatch a command. See the command and function discussions in this chapter for information on the appropriate parameters.

## Structures for the Printer Device

The IOStdReq, IOPrtCmdReq, and IODRPReq structures are used to define Printer device commands to be dispatched to the internal routines of the predefined Printer device library. The IOStdReq structure is presented in Chapter 2. Figure 11.3 shows the IOPrtCmdReq and IODRPReq structures.

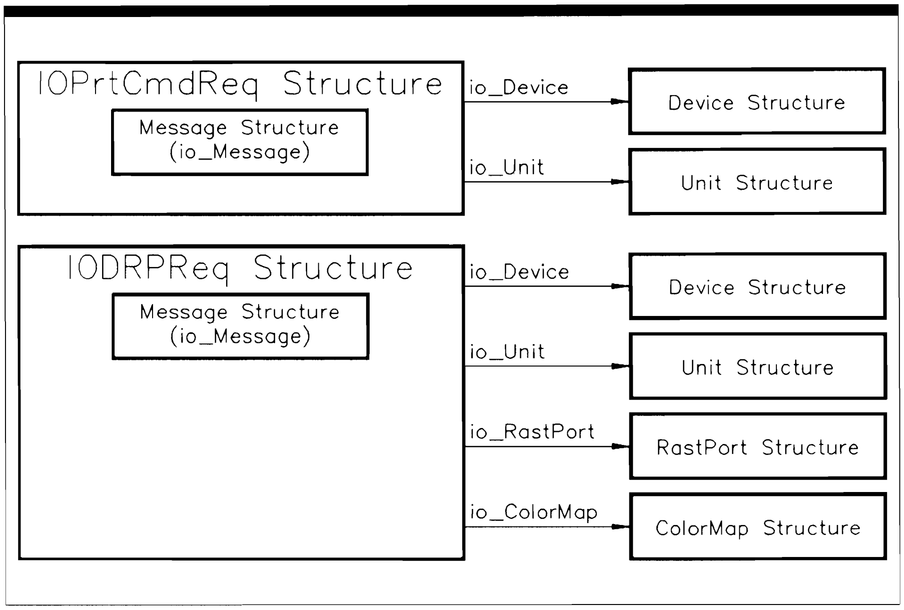


**Figure 11.3:** *Printer Device Structures*

## The IOPrtCmdReq Structure

The IOPrtCmdReq structure is used only with PRD_PRTCOMMAND. It is defined as follows:

```
struct IOPrtCmdReq {
   struct Message io_Message;
   struct Device *io_Device;
   struct Unit *io_Unit;
   UWORD io_Command;
   UBYTE io_Flags;
   BYTE io_Error;
   UWORD io_PrtCommand;
   UBYTE io_Parm0;
   UBYTE io_Parm1;
   UBYTE io_Parm2;
   UBYTE io_Parm3;
};
```

The parameters in the IOPrtCmdReq structure have these meanings:

- io_Message. This is the name of a Message substructure. It contains the mn_ReplyPort parameter, which specifies the task reply port that will receive the PRD_PRTCOMMAND reply when the Printer device internal routines have finished processing it.

- io_Device. This points to a Device structure that manages the Printer device internal routines when the device has been opened with OpenDevice.
- io_Unit. This points to a Unit structure containing the MsgPort substructure that represents the device-unit request queue.
- io_Command. This is the command you want the Printer device to execute—PRD_PRTCOMMAND.
- io_Flags. This is a set of flag parameters; the Printer device does not use them.
- io_Error. This contains the error messages returned by the Printer device.
- io_PrtCommand. This is the Printer-specific control-code command you want the Printer device internal routines to execute. It always contains an escape character as its first character.
- io_Parm0, io_Parm1, io_Parm2, and io_Parm3. These are the first, second, third, and fourth parameters in the control-code command used with PRD_PRTCOMMAND.

For more information on this structure, see the Printer.h INCLUDE file and the PRD_PRTCOMMAND discussion.

## The IODRPReq Structure

The IODRPReq structure is used only with the PRD_DUMPRPORT command. It is defined as follows:

```
struct IODRPReq {
   struct Message io_Message;
   struct Device *io_Device;
   struct Unit *io_Unit;
   UWORD io_Command;
   UBYTE io_Flags;
   BYTE io_Error;
   struct RastPort *io_RastPort;
   struct ColorMap *io_ColorMap;
   ULONG io_Modes;
   UWORD io_SrcX;
   UWORD io_SrcY;
   UWORD io_SrcWidth;
   UWORD io_SrcHeight;
   LONG io_DestCols;
   LONG io_DestRows;
   UWORD io_Special;
};
```

The io_Message, io_Device, io_Unit, io_Flags, and io_Error parameters have the same meanings as they do for the IOPrtCmdReq structure. The other IODRPReq parameters have the following meanings:

- io_Command. This is the command you want the Printer device to execute—PRD_DUMPRPORT.
- io_RastPort. This points to a Graphics library RastPort structure that represents the bitmap containing information you want to send to the printer.
- io_ColorMap. This points to a Graphics library ColorMap structure that is used to define the colors for printing the bitmap information.
- io_Modes. This is a set of graphics viewport modes used to print the bitmap information. It is the same as the Graphics library ViewPort structure Modes parameter.
- io_SrcX and io_SrcY. These are the X and Y origins of the information in the raster bitmap coordinate system.
- io_SrcWidth and io_SrcHeight. These are the width and height of the bitmap information to be sent to the printer.
- io_DestCols. This is the width of the bitmap information to be sent to the printer.
- io_DestRows. This is the height of the bitmap information to be sent to the printer.
- io_Special. This is a set of special option flags that control the interpretation of the io_DestCols and io_DestRows parameters.

See the discussion of PRD_DUMPRPORT and the Printer.h INCLUDE file for more about this structure.

## USE OF FUNCTIONS

## *CloseDevice*

### Syntax of Function Call

```
CloseDevice (printerIO)
                A1
```

### Purpose of Function

This function closes access to Printer device unit 0. If it is the last CloseDevice function call in a task and the Serial and Parallel devices have also been closed by CloseDevice calls in the task, the Serial and Parallel devices will also be closed. When CloseDevice returns, the current task cannot use the Printer device until it executes another OpenDevice function call.

CloseDevice sets the PrinterIO union IOStdReq, IOPrtCmdReq, or IODRPReq substructure io_Device and io_Unit parameters to $-1$. The PrinterIO union cannot be used again until these parameters are reinitialized by an OpenDevice call. CloseDevice also decrements the Device structure lib_OpenCnt and the Unit structure unit_OpenCnt parameters by 1.

### Inputs to Function

| | |
|---|---|
| **printerIO** | A pointer to a PrinterIO union |

### Discussion

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

When a task is done with its Printer device operations, it should close the device with a call to CloseDevice. This frees RAM that might be needed by the system for other tasks. Another task can then open, use, and close the Printer device; the sequence can be repeated again and again in a C language program.

Keep in mind that the Printer device opens either the Serial or the Parallel device indirectly; therefore, a CloseDevice function call for the Printer device can affect these devices.

## *OpenDevice*

### Syntax of Function Call

```
error = OpenDevice ("printer.device", 0,  PrinterIO, 0)
D0                  A0               D0  A1         D1
```

## Purpose of Function

This function opens access to the internal routines of Printer device unit 0. An OpenDevice call for the Printer device will automatically open the Serial or Parallel device, depending on the type of printer currently selected on the Workbench Preferences screen.

Once OpenDevice has opened the Printer device, it initializes printer device internal parameters. It also increments the Device structure lib_OpenCnt parameter and the Unit structure unit_OpenCnt parameter. OpenDevice requires a properly initialized reply port with a task signal bit allocated if the calling task wants to be signaled when the Printer device routines reply. The results of function execution are found in the following parameters:

- io_Device. This points to a Device structure that manages the Printer device internal routines for unit 0 once it is opened.
- io_Unit. This points to a Unit structure that defines and manages a MsgPort structure for Printer device unit 0. The MsgPort structure represents the device request queue.
- io_Error. A 0 value for io_Error indicates that the request was successful. IOERR_OPENFAIL indicates that the Printer device could not be opened; this usually means there is not enough available memory.

## Inputs to Function

| | |
|---|---|
| **"printer.device"** | A pointer to a null-terminated string representing the name of the Printer device |
| **unitNumber** | Always 0 |
| **PrinterIO** | A pointer to a PrinterIO union |
| **0** | Indicates that the flags argument is ignored |

## Preparation of the PrinterIO Union

Initialize the IOStdReq substructure mn_ReplyPort parameter to point to the MsgPort structure representing the desired task reply port. Set all other IOStdReq substructure parameters to 0, or copy them from the IOStdReq structure of a previous OpenDevice call. Initialize io_Command to 0, or initialize it to PRD_DUMPRPORT, PRD_PRTCOMMAND, PRD_RAWWRITE, or CMD_WRITE if you want to open the Printer device and then dispatch a command.

If the CreateExtIO function is used to allocate and initialize the PrinterIO union (see Chapter 2), it will return a pointer to an IOStdReq structure automatically. Therefore, for the Printer device, the result returned by CreateExtIO must be typecast into a pointer to a PrinterIO union.

### Discussion

OpenDevice can be called to open the Printer device and dispatch a PRD_DUMPR-PORT, PRD_PRTCOMMAND, PRD_RAWWRITE, or CMD_WRITE command to the Printer device internal routines. Once a task has opened the Printer device, it can dispatch a series of these and other Printer device commands to send information to the printer. When a task has finished its Printer device processing, it should close the device with CloseDevice.

Most IOPrtCmdReq and IODRPReq substructure parameters can be initialized to represent commands after the Printer device has been opened. Parameters that are not explicitly initialized in a PrinterIO substructure will retain their previous values or be initialized to default values. If a task needs to use values other than the defaults for these parameters, it should initialize them after OpenDevice returns.

## STANDARD DEVICE COMMANDS

## *CMD_FLUSH*

### Purpose of Command

CMD_FLUSH aborts all active and queued I/O requests. It is an immediate-mode command, allows QuickIO, and always replies to the task reply-port queue if the IOF-_QUICK bit is not set. The results of command execution are found in io_Error, where 0 indicates that the command was successful. IOERR_NOCMD indicates that the task specified io_Command incorrectly.

### Preparation of the PrinterIO Union

Initialize the IOStdReq substructure mn_ReplyPort parameter to point to the MsgPort structure representing the desired task reply port. Initialize its io_Device and io_Unit parameters to point to the Device and Unit structures that manage Printer device unit 0; these can always be copied from the IOStdReq substructure initialized by the OpenDevice function call. Also initialize io_Command to CMD_FLUSH. Set io_Flags to IOF-_QUICK for QuickIO; otherwise, set it to 0.

### Discussion

CMD_FLUSH aborts all active and pending PRD_DUMPRPORT, PRD_PRTCOM-MAND, PRD_RAWWRITE, and CMD_WRITE requests from the unit 0 device request queue. Because CMD_FLUSH is destructive, you should use it only if you want to restore the system to some known state with an empty Printer device request queue. CMD_FLUSH does not affect the state of any task reply-port queue except for the addition of the replied aborted I/O requests. Each aborted I/O request will have its io_Error parameter set to IOERR_ABORTED.

## CMD_RESET

### Purpose of Command

CMD_RESET resets Printer device unit 0 to its boot-up state as if it were just initialized by an OpenDevice call—all Printer device internal parameters are set to their default values. CMD_RESET allows QuickIO and always replies to the task reply-port queue if the IOF_QUICK bit is not set. The results of command execution are found in io_Error, where 0 indicates that the command was successful.

### Preparation of the PrinterIO Union

Initialize the IOStdReq substructure mn_ReplyPort parameter to point to the MsgPort structure representing the desired task reply port. Initialize its io_Device and io_Unit parameters to point to the Device and Unit structures that manage Printer device unit 0. These can always be copied from the IOStdReq substructure initialized by the OpenDevice function call. Also set io_Command to CMD_RESET. Set io_Flags to IOF-_QUICK for QuickIO; otherwise, set it to 0.

### Discussion

CMD_RESET does not destroy the IOStdReq substructure io_Device and io_Unit parameters provided by OpenDevice when the Printer device was opened; a task can still use copies of these in subsequent I/O requests.

## CMD_START

### Purpose of Command

CMD_START immediately restarts commands to Printer device unit 0 if they were stopped by CMD_STOP, including any command that was stopped in the middle of its activity and the first command at the top of the device request queue when CMD_STOP is dispatched. CMD_START is an immediate-mode command, allows QuickIO, and always replies to the task reply-port queue if the IOF_QUICK bit is not set. The results of command execution are found in io_Error, where 0 indicates that the command was successful. IOERR_NOCMD indicates that the task specified io_Command incorrectly.

### Preparation of the PrinterIO Union

Initialize the IOStdReq substructure mn_ReplyPort parameter to point to the MsgPort structure representing the desired task reply port. Initialize its io_Device and io_Unit parameters to point to the Device and Unit structures that manage Printer device unit 0; these can always be copied from the IOStdReq substructure initialized by the OpenDevice function call. Also initialize io_Command to CMD_START. Set io_Flags to IOF-_QUICK for QuickIO; otherwise, set it to 0.

### Discussion

CMD_START is similar to the Ctrl-Q command used to restart screen output on most computers. It immediately restarts execution of a command previously stopped by CMD-_STOP, just as Ctrl-Q restarts screen output previously stopped with Ctrl-S. CMD-_START also starts the execution of other queued I/O requests, just as Ctrl-Q displays additional files on the screen if the user has typed file-display commands.

## *CMD_STOP*

### Purpose of Command

CMD_STOP stops command execution immediately. It also prevents the Printer device internal routines from executing queued requests. Once unit 0 is stopped by CMD-_STOP, the system automatically queues requests dispatched to it until CMD_START restarts it. Once CMD_STOP executes, only CMD_START and CMD_FLUSH requests can be dispatched to the Printer device.

CMD_STOP is an immediate mode command, supports QuickIO, and always replies to the task reply-port queue if the IOF_QUICK bit is not set. The results of command execution are found in io_Error, where a 0 value indicates that the command was successful. IOERR_NOCMD indicates that the task specified io_Command incorrectly.

## Preparation of the PrinterIO Union

Initialize the IOStdReq substructure mn_ReplyPort parameter to point to the MsgPort structure representing the desired task reply port. Initialize its io_Device and io_Unit parameters to point to the Device and Unit structures that manage Printer device unit 0; these can be copied from the IOStdReq substructure initialized by the OpenDevice function call. Also set io_Command to CMD_STOP. Set io_Flags to IOF_QUICK for QuickIO; otherwise, set it to 0.

## Discussion

CMD_STOP stops the execution of a PRD_DUMPRPORT, PRD_PRTCOMMAND, PRD_RAWWRITE, or CMD_WRITE command immediately. It is similar to the Ctrl-S command used for screen output on most computers—it stops an executing command at the earliest possible opportunity. I/O requests in the Printer device request queue will not be executed until a CMD_START command is dispatched to unit 0.

## *CMD_WRITE*

The CMD_WRITE command causes a stream of standard ANSI 3.64 characters to be written from a task-defined buffer to the printer. The Printer device uses the Parallel or the Serial device indirectly to produce the printing. Each character in the character stream can represent a printer control code, which can initialize or reset the printer, turn italic type on and off, and so on.

The characters are written from a task-defined buffer that contains the number of characters specified by the PrinterIO union IOStdReq substructure io_Length parameter. If −1 is specified for io_Length, however, the CMD_WRITE command will write characters until an EOF character is written. A CMD_WRITE command can be terminated early if a write error occurs or if an EOF condition defined by a hexadecimal 0x00 character is encountered.

Because CMD_WRITE does not support QuickIO, its I/O request structure is always replied to the sending task's reply-port queue. The results of command execution are found in io_Error, where a 0 value indicates that the command was successful. PDERR_INTERNALMEMORY indicates that there was not enough RAM for the necessary Printer device internal routines when CMD_WRITE was dispatched. PDERR_BUFFERMEMORY indicates that there was not enough RAM for the task-defined buffer. IOERR_NOCMD indicates that the task specified io_Command incorrectly, and IOERR_BADLENGTH indicates that the task specified io_Length incorrectly.

## Preparation of the PrinterIO Union

Initialize the IOStdReq substructure mn_ReplyPort parameter to point to the MsgPort structure representing the desired task reply port. Initialize its io_Device and io_Unit parameters to point to the Device and Unit structures that manage Printer device unit 0; these can always be copied from the IOStdReq substructure initialized by the OpenDevice function call. Set io_Command to CMD_WRITE, and set io_Flags to 0. Set io_Length to the number of characters to be sent to the serial or parallel port, or set it to −1 to tell the task to write characters until the hexadecimal 0x00 EOF character is written to the printer port from the task-defined write buffer. Also set io_Data to point to the task's write buffer.

## Discussion

The Printer device, like the Console device, maps (translates) the standard 7-bit ANSI 3.64 control-code characters to those of a connected printer. The Printer device internal routines perform the translation for the printer selected in the Workbench Preferences screen. (Not all printers, however, support all the standard control-code characters.)

The CMD_WRITE command provides a way to transmit a series of control-code commands to a printer by dispatching one Printer device command. Note that you use PRD_PRTCOMMAND to transmit one control-code command at a time.

## DEVICE-SPECIFIC COMMANDS

## *PRD_DUMPRPORT*

## Purpose of Command

The PRD_DUMPRPORT command causes all or some of a raster bitmap to be printed. Several of the Graphics library RastPort, ViewPort, ColorMap, and ViewPort structure parameters are used to specify how the raster bitmap should be printed. In addition, IODRPReq structure parameters are used to control the size and other characteristics of the printed result.

PRD_DUMPRPORT supports QuickIO and always replies to the task reply-port queue if the IOF_QUICK bit is not set. The results of command execution are found in io_Error, where a 0 value indicates that the command was successful. Other error values are as follows:

- PDERR_NOTGRAPHICS indicates that the specified printer is not a graphics printer and cannot print a raster bitmap.

- PDERR_BADDIMENSION indicates that the dimensions specified in the IODRPReq structure are not valid, are in conflict, or are inconsistent.
- PDERR_DIMENSIONOVFLOW indicates that the dimensions specified in the IODRPReq structure caused a dimension overflow.
- IOERR_NOCMD indicates that the task specified io_Command incorrectly.

## Preparation of the PrinterIO Union

Initialize the IOStdReq substructure mn_ReplyPort parameter to point to the MsgPort structure representing the desired task reply port. Initialize its io_Device and io_Unit parameters to point to the Device and Unit structures that manage Printer device unit 0; these can always be copied from the IOStdReq substructure initialized by the OpenDevice function call. Set io_Command to PRD_DUMPRPORT. Set io_Flags to IOF_QUICK for QuickIO; otherwise, set it to 0.

Also initialize the following parameters in the PrinterIO union IODRPReq substructure. See Chapter 2 of Volume I for a detailed discussion of the RastPort and BitMap structure parameters included here.

- io_RastPort. Initialize this parameter to point to a Graphics library RastPort structure that will control the printing.
- io_ColorMap. Initialize this parameter to point to a Graphics library ColorMap structure that will control the printing colors.
- io_Modes. Initialize this parameter to the Graphics library ViewPort structure Modes parameter. This will control the drawing modes, which determine how the raster bitmap bits are interpreted during the printing process.
- io_SrcX. Initialize this parameter to the X offset from the raster bitmap origin specified by the RastPort structure.
- io_SrcY. Initialize this parameter to the Y offset from the raster bitmap origin specified by the RastPort structure.
- io_SrcWidth. Initialize this parameter to the number of X-direction pixels in the portion of the raster bitmap that the task should print.
- io_SrcHeight. Initialize this parameter to the number of Y-direction pixels in the portion of the raster bitmap that the task should print.
- io_DestCols. Set this parameter to the number of pixel columns to be printed (see io_Special).
- io_DestRows. Set this parameter to the number of pixel rows to be printed (see io_Special).

- io_Special. Initialize this parameter to one or a combination of 13 possible values. Set it to SPECIAL_MILCOLS and SPECIAL_MILROWS if you want the io_DestCols and io_DestRows parameter values to represent multiples of one one-thousandth of an inch on the printed page. Set it to SPECIAL_FULLCOLS and SPECIAL_FULLROWS if you want the io_DestCols and io_DestRows parameters to be ignored; the raster bitmap will then be printed within the maximum printer-page dimensions determined by the printer's physical limits or the system configuration. Set it to SPECIAL_FRACCOLS and SPECIAL_FRACROWS if you want the io_DestCols and io_DestRows parameters to represent a fraction of the maximum printer-page dimension for each printing direction; in this case, the parameters are specified as longword (32-bit) binary fractions of the maximum printer-page dimensions. Set the ASPECT bit if you want one of the specified io_DestCols and io_DestRows parameters to be reduced to preserve the aspect ratio of the printed result. This option also works with the six previous options.

  Set SPECIAL_DENSITYMASK, which is a bit mask for DENSITY1 through DENSITY4, if you want to specify a printing density. Set one of DENSITY1 through DENSITY4 to the specific density that your printer supports. Set the CENTER bit if you want the printout centered on the page (if your printer supports this).

  If all of the io_Special bits are cleared, the io_DestRows and io_DestCols parameters will be interpreted as the number of pixels in the current resolution of the raster bitmap. Unexpected results can occur when these bits are specified as 0 or are initialized inconsistently.

## Discussion

PRD_DUMPRPORT allows a task to print all or part of a raster bitmap to the printer. It can produce either a grey-scale printout or a color printout.

Recall from Volume I (Chapter 2) that a raster bitmap is specified by a combination of the Graphics library RastPort, ViewPort, ColorMap, and BitMap structures (among others). Just as a task can use the parameters to these structures to define how a raster bitmap will be displayed on the screen, it can use them to define the printing of some or all of a raster bitmap on paper.

As an example, if you are making a document to illustrate one of your Intuition programs, you may want to print a Workbench screen containing one of your program's windows. You can do this by opening the Intuition library (with an OpenLibrary function call); initializing a NewWindow structure; opening the window (with an OpenWindow function call); copying a pointer to the Workbench screen ViewPort structure; and copying the Workbench Screen structure RastPort, ViewPort, ColorMap, and BitMap parameters. You can use these parameters to define those in the IODRPReq structure used to define the PRD_DUMPRPORT command. PRD_DUMPRPORT will then print the Workbench screen that shows your Intuition window.

## PRD_PRTCOMMAND

### Purpose of Command

The PRD_PRTCOMMAND command causes a stream of at most four characters defining a printer control-code command to be written from a set of four PrinterIO union IOPrtCmdReq substructure parameters to the printer. The printer control-code command is an ANSI 3.64 command name together with as many as four characters to define it. Once the ANSI 3.64 representation is translated by the Printer device internal routines, it defines a printer-specific command. The Printer device uses either the Parallel or Serial device indirectly to produce the printing. A PRD_PRTCOMMAND command can be terminated early if a write error occurs.

Because PRD_PRTCOMMAND does not support QuickIO, its I/O request structure always replies to the sending task's reply-port queue. The results of command execution are found in io_Error, where a 0 value indicates that the command was successful. PDERR_INTERNALMEMORY indicates that there was not enough RAM for the Printer device internal routines when PRD_PRTCOMMAND was dispatched. IOERR_NOCMD indicates that the task specified io_Command incorrectly.

### Preparation of the PrinterIO Union

Initialize the IOStdReq substructure mn_ReplyPort parameter to point to the MsgPort structure representing the desired task reply port. Initialize its io_Device and io_Unit parameters to point to the Device and Unit structures that manage Printer device unit 0. These can always be copied from the IOStdReq substructure initialized by the OpenDevice function call. Set io_Command to PRD_PRTCOMMAND. Set io_Flags to 0.

The four-character control-code command is specified in io_Parm0, io_Parm1, io_Parm2, and io_Parm3. Some or all of these parameters may need to be initialized to 0; see the *ROM Kernel Manual* for a definition of each printer command and its associated parameters. All other IOStdReq substructure parameters should be initialized to 0.

### Discussion

PRD_PRTCOMMAND allows a task to send a single control-code command to a printer connected to the serial or parallel port. As many as four bytes of printer control code are transferred; they are interpreted as a unit to define the command. The Printer device internal routines map the standard 7-bit ANSI 3.64 control-code characters to those for the connected printer. However, keep in mind that not all printers support the standard control-code characters and the printer commands they define.

PRD_PRTCOMMAND sends a single control-code command to a printer. On the other hand, the CMD_WRITE command transmits a series of control-code commands to the printer. Use whichever command best suits your purposes.

## *PRD_RAWWRITE*

### Purpose of Command

The PRD_RAWWRITE command causes a stream of raw characters to be written from a task-defined buffer to the printer. These characters usually consist of escape sequences. With this command, the Printer device routines do not translate the raw characters; they use the Parallel or Serial device indirectly to produce the printing.

Each character in the raw character stream can represent a printer control-code command; the printer will interpret the control code correctly. The raw characters are written from a task-defined buffer that contains the number of characters specified by the PrinterIO union IOStdReq substructure io_Length parameter. In contrast to the CMD_WRITE command, the PRD_RAWWRITE command does not allow io_Length to be specified as −1; it does not work with EOF characters. A PRD_RAWWRITE command can be terminated early if a write error occurs.

PRD_RAWWRITE allows QuickIO and always replies to the task reply-port queue if the IOF_QUICK bit is not set. The results of command execution are found in io_Error, where 0 indicates that the command was successful. PDERR_INTERNALMEMORY indicates that there was not enough RAM for the Printer device internal routines when PRD_RAWWRITE was dispatched. IOERR_NOCMD indicates that the task specified io_Command incorrectly, and IOERR_BADLENGTH indicates that the task specified io_Length incorrectly.

### Preparation of the PrinterIO Union

Initialize the IOStdReq substructure mn_ReplyPort parameter to point to the MsgPort structure representing the desired task reply port. Initialize its io_Device and io_Unit parameters to point to the Device and Unit structures that manage Printer device unit 0; these can always be copied from the IOStdReq substructure initialized by the OpenDevice function call. Set io_Command to PRD_RAWWRITE.

Initialize io_Flags to IOF_QUICK for QuickIO; otherwise, set it to 0. Set io_Length to the number of characters to be sent to the serial or parallel port from the task-defined write buffer, and set io_Data to point to the task-defined write buffer.

## Discussion

The PRD_RAWWRITE command provides a way to transmit a series of control-code commands to a printer by dispatching one Printer device command. The PRD_PRT-COMMAND command, on the other hand, can transmit only one control-code command at a time. Use whichever of these commands best suits your purposes.

# 12

# The Clipboard Device

## Introduction

The Clipboard device manages a set of clipboard files, one for each open Clipboard device unit. A task can read and write information into the clipboard files, which allows it to cut and paste (read and write) information in a file as necessary. Clipboard files also allow the exchange of data among a group of related tasks.

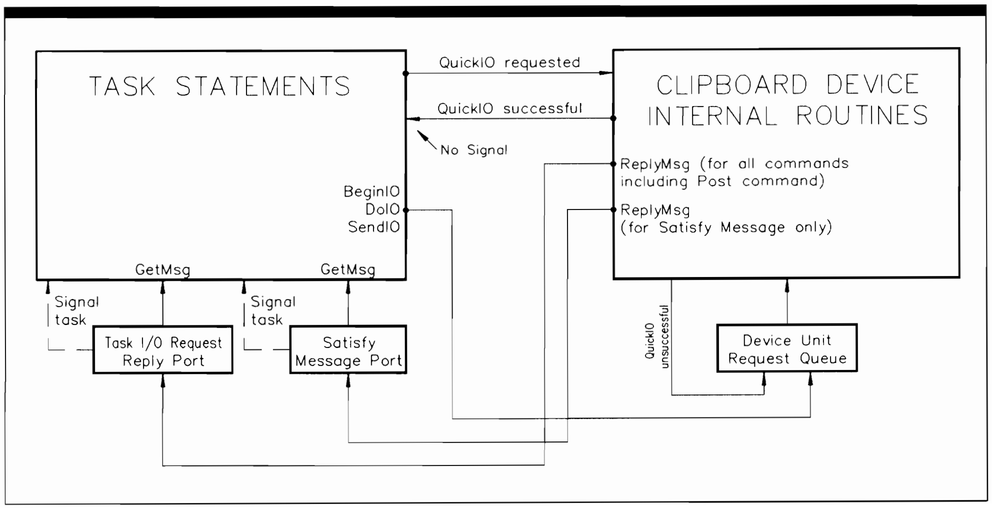
The Clipboard device is programmed with several Exec functions, as well as four standard device and three device-specific commands. This device is disk-resident; it is loaded into WCS memory from disk when OpenDevice executes.

## Operation of the Clipboard Device

Figure 12.1 illustrates the general operation of the Clipboard device, which consists of one or more units, each of which can queue commands for processing by the Clipboard device internal routines. A new clipboard file is created by opening a new Clipboard device unit with an OpenDevice function call. To simplify the discussion, only one Clipboard device unit is shown in Figure 12.1; however, a task can manage any number of clipboard files.

The device request queue for a Clipboard device unit is managed by a Unit structure, and a pointer to it is specified by OpenDevice when it returns. A MsgPort structure represents the device-unit request queue, which receives the queued commands coming from the task to the unit's internal routines. Command requests are queued in FIFO (first in, first out) order.

The task itself should create and establish signals for two message ports using two calls to the Exec-support library CreatePort function. The first message port is used as a



TASK STATEMENTS
QuickIO requested
QuickIO successful
No Signal
CLIPBOARD DEVICE INTERNAL ROUTINES
ReplyMsg (for all commands including Post command)
ReplyMsg (for Satisfy Message only)
BeginIO
DoIO
SendIO
GetMsg
GetMsg
Signal task
Signal task
Task I/O Request Reply Port
Satisfy Message Port
QuickIO unsuccessful
Device Unit Request Queue

**Figure 12.1:** *Operation of the Clipboard Device*

task reply-port queue. The Clipboard device internal routines use the Exec ReplyMsg function to send replies to this queue.

The second port, called the satisfy message port, is used as a task reply-port queue for the SatisfyMsg structure messages. The Clipboard device internal routines send satisfy messages to this port automatically when they determine that a prior CBD_POST command must be satisfied by a CMD_WRITE command. Since the satisfy message port was created with a signal to it established, the task is signaled of the message's arrival automatically. The task then dispatches a CMD_WRITE command to satisfy the earlier CBD_POST request and uses the GetMsg or Remove function to remove the message from the port so that it is empty when the next satisfy message arrives.

A satisfy message is automatically created and replied by the Clipboard device internal routines—your task is not responsible for allocating and initializing it. The message simply tells the task that the Clipboard device internal routines require that it dispatch a CMD_WRITE command in order to add new data to the current clipboard file. In this sense, the CBD_POST command acts as a deferred CMD_WRITE command.

## Ordinal Clip Identifiers

One of the most difficult aspects of working with Clipboard device functions and commands is understanding how ordinal clip identifiers work. *Ordinal clip identifier* is another name for the IOClipReq structure io_ClipID parameter. It identifies the order in which CMD_READ, CMD_WRITE, and CBD_POST commands have been dispatched by a task.

The Clipboard device maintains a read clip identifier and a write clip identifier. Both of these consist of consecutive integers that start at 0. Each task that works with the Clipboard device must have its own read and write clip-identifier lists. The Clipboard device internal routines work with the IOClipReq structure io_ClipID parameter to determine the sequence of clips defined by CMD_READ, CMD_WRITE, and CBD_POST commands.

A task should initialize the read or write clip identifier to 0 for the first CMD_READ or CMD_WRITE command. Thereafter, the Clipboard device internal routines will update the io_ClipID parameter to reflect any new CMD_READ or CMD_WRITE commands. The task can determine the clip identifier of the most recently dispatched CMD_READ or CMD_WRITE command by using the CBD_CURRENTREADID or CBD_CURRENTWRITEID command. It uses this information to determine if the CBD_POST data it holds should be written to the current clipboard file, or if it is obsolete and need not be sent.

## Sequential Read-Write Operations for a Clipboard File

Figure 12.2 shows how a task performs a set of sequential read and write operations for the current clipboard file. The following discussion examines sequential CMD_READ commands; the same procedure, in reverse, applies to sequential CMD_WRITE commands.

A sequential read is a series of consecutive CMD_READ commands that a task dispatches in order to read different segments of the current clipboard file into a task-defined buffer. The task could have placed the data bytes in the clipboard file using single or sequential CMD_WRITE commands. The figure shows how a task can read the contiguous

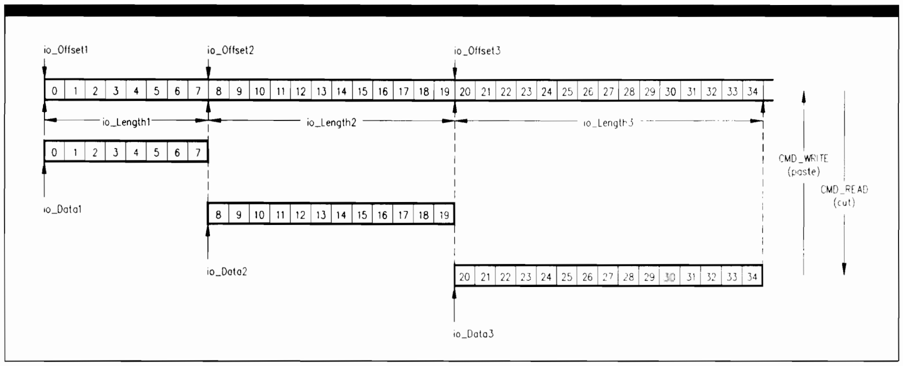


**Figure 12.2:** *Sequential Read–Write Operations for a Clipboard File*

segments of the clipboard file into a set of task-defined buffers; the segments are 8, 12, and 14 bytes long.

A task defines a series of sequential reads by using the io_Data, io_Length, and io_Offset parameters. As an example, assume that you want your task to read the current clipboard file in Figure 12.2 sequentially, starting at the beginning of the file. Here is the procedure you would follow:

1. Define an IOClipReq structure to represent the first CMD_READ command. Initialize its io_Data parameter to point to the first task-defined buffer (1). This defines the RAM location to which the clipboard bytes should be sent. Initialize io_Length to 1, the number of bytes you want transferred to the task-defined buffer. Initialize io_Offset to 0 to represent the first byte of the current clipboard file. Dispatch this CMD_READ command with BeginIO, DoIO, or SendIO. When CMD_READ completes execution, the Clipboard device internal routines assign a value to io_Offset to reflect that the clipboard-file byte pointer is positioned io_Length1 bytes into the file. This is the starting position for the next CMD_READ command. The Clipboard device internal routines also assign a value to the IOClipReq structure io_ClipID parameter. The second and third CMD_READ commands will use the same IOClipReq structure and will not alter the io_ClipID parameter; in this way, the internal routines can recognize and keep track of a set of related CMD_READ commands for the same clipboard file.

2. Define an IOClipReq structure to represent the second CMD_READ command. Initialize io_Data to 2, which indicates that it points to the second task-defined buffer. You can use the same task-defined buffer you used in step 1 if the task has already processed the clipboard bytes read by the first CMD_READ command. Initialize io_Length to 2, the number of bytes you want transferred to the task-defined buffer. Do not reinitialize io_Offset; it has been updated automatically by the Clipboard device and is positioned io_Length1 bytes into the current clipboard file. Use BeginIO, DoIO, or SendIO to dispatch the command. When CMD_READ completes execution, the clipboard-file byte pointer io_Offset parameter will be positioned (io_Length1 + io_Length2) bytes into the clipboard file, which is the starting position for the third CMD_READ command.

3. Define an IOClipReq structure to represent the third CMD_READ command. Initialize io_Data to 3, the third task-defined buffer. You can use the same task-defined buffer if the task has processed the clipboard bytes read by the first and second commands. Initialize io_Length to 3, the number of bytes you want transferred to the buffer. Once again, do not initialize io_Offset; it is updated automatically to reflect the first and second CMD_READ commands. Dispatch the command. When CMD_READ completes execution, the clipboard-file byte pointer io_Offset parameter is positioned (io_Length1 + io_Length 2 + io_Length3) bytes into the current clipboard file.

Each CMD_READ command returns the actual number of bytes read in the IOClipReq structure io_Actual parameter. The task can examine this parameter when the CMD_READ command is replied, compare it to the input value of the io_Length parameter, and thereby verify that CMD_READ executed properly.

This programming sequence also works for a set of sequential CMD_WRITE commands. However, for sequential CMD_WRITE commands, you are transferring data bytes from the task-defined buffers into contiguous areas of the clipboard file. Each new area is adjacent to the previous area.

## Clipboard Device Commands

The Clipboard device has three device-specific and four standard device commands. All of these support queued I/O, but only CMD_RESET supports QuickIO. None of the commands support automatic immediate-mode operation. All seven commands affect the IOClipReq structure io_Error parameter. Five commands also affect the IOClipReq structure io_ClipID parameter, and two affect the io_Actual and io_Offset parameters.

### Sending Commands to the Clipboard Device

Figure 12.3 illustrates how commands are dispatched to the Clipboard device. The lines with arrows represent the parameters you initialize, as well as those returned by the Clipboard device internal routines.

The Clipboard device programming process consists of three phases:

1. Structure preparation. You have complete control over this phase; here, you initialize parameters in the IOStdReq and IOClipReq structures before dispatching a command to the Clipboard device routines. These parameters include those required by most devices, as well as the io_ClipID and io_Offset parameters for some commands. The choice of parameters to initialize depends on the specific command you plan to dispatch. Taken together, these parameters provide an information path to the data needed by the Clipboard device in order for it to process the command.

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

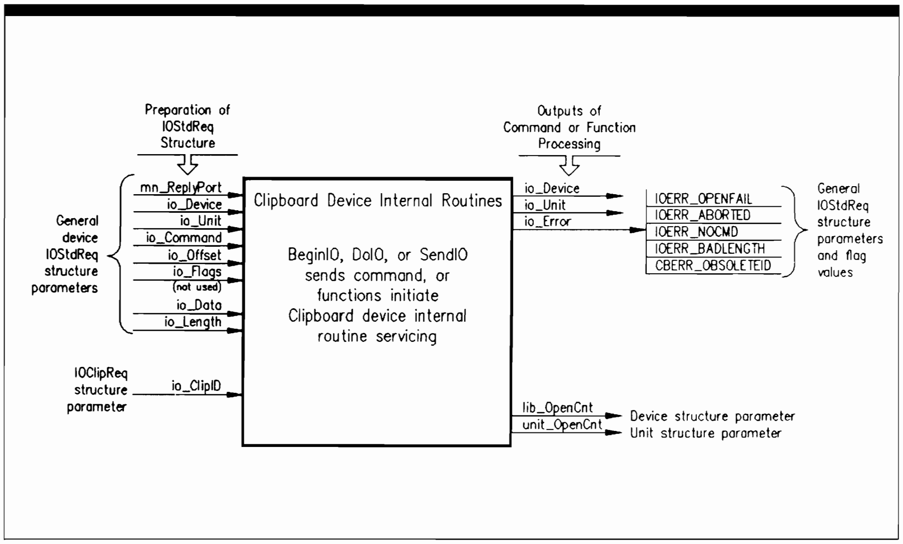

Preparation of IOStdReq Structure
Outputs of Command or Function Processing
General device IOStdReq structure parameters
mn_ReplyPort
io_Device
io_Unit
io_Command
io_Offset
io_Flags
(not used)
io_Data
io_Length
IOClipReq structure parameter
io_ClipID
Clipboard Device Internal Routines
BeginIO, DoIO, or SendIO sends command, or functions initiate Clipboard device internal routine servicing
io_Device
io_Unit
io_Error
IOERR_OPENFAIL
IOERR_ABORTED
IOERR_NOCMD
IOERR_BADLENGTH
CBERR_OBSOLETEID
General IOStdReq structure parameters and flag values
lib_OpenCnt
unit_OpenCnt
Device structure parameter
Unit structure parameter

**Figure 12.3:** *Clipboard Device Command and Function Processing*

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

Several Clipboard device commands provide output parameters, as shown on the right side of Figure 12.3. Both the CMD_READ and CMD_WRITE commands return values for the IOClipReq structure io_Error, io_Actual, io_Offset, and io_ClipID parameters.

Figure 12.3 also shows the parameters that play a part in Clipboard device function setup and processing. OpenDevice and CloseDevice both affect the Unit structure unit_OpenCnt parameter and the Device structure lib_OpenCnt parameter associated with open units; OpenDevice also affects the io_Error parameter.

## Structures for the Clipboard Device

The Clipboard device deals with three structures: ClipboardUnitPartial, IOClipReq, and SatisfyMsg. The ClipboardUnitPartial structure is used to maintain a list of Clipboard device units in the system. Each OpenDevice function call for a new Clipboard device unit adds a member to this list. The IOClipReq structure is used to formulate and dispatch all command requests to the Clipboard device internal routines. The SatisfyMsg structure is used by the Clipboard device internal routines to dispatch a satisfy message to a task. When the task receives this message, it should immediately dispatch a

CMD_WRITE command to satisfy the original CBD_POST command. These three structures are shown in Figure 12.4.

## The ClipboardUnitPartial Structure

The ClipboardUnitPartial structure is defined as follows:

```
struct ClipboardUnitPartial {
    struct Node cu_Node;
    ULONG cu_UnitNum;
};
```

Its parameters have the following meanings:

- cu_Node. This is the name of a Node substructure that places a new Clipboard device unit on the system clipboard list.
- cu_UnitNum. This is the unit number of the Clipboard device unit. A new unit is assigned a unit number by the Clipboard device internal routines. The Unit structure parameters (unit_MsgPort, unit_Flags, and unit_OpenCnt) are controlled by the Clipboard device internal routines; they are said to be private to the Clipboard device.

## The IOClipReq Structure

The IOClipReq structure is defined as follows:

```
struct IOClipReq {
    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;
    STRPTR io_Data;
    ULONG io_Offset;
    LONG io_ClipID;
};
```

Its parameters have these meanings:

- io_Message. This is the name of a Message substructure used to represent the I/O request dispatched to the device-unit request queue by a task. The Message structure mn_ReplyPort parameter points to the MsgPort structure representing the task reply-port queue.

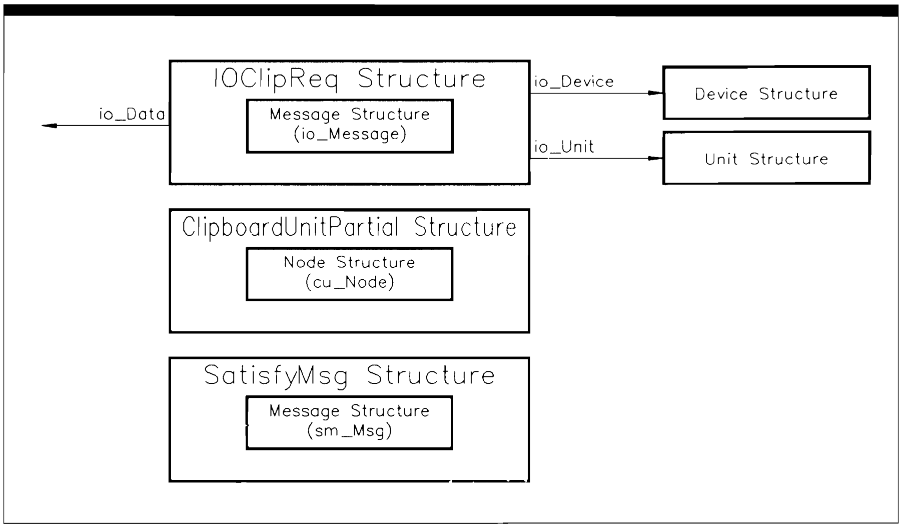


**Figure 12.4:** *Clipboard Device Structures*

- io_Device. This points to a Device structure representing the Clipboard device unit. It manages the Clipboard device internal routines.
- io_Unit. This points to a Unit structure for the Clipboard device unit. The unit_MsgPort parameter in the Unit structure points to a MsgPort structure representing the device-unit request queue.
- io_Command. This is the command that you want the Clipboard device internal routines to execute.
- io_Flags. This parameter includes the IOF_QUICK and IOF_SATISFY flag parameter bits.
- io_Error. This parameter contains Clipboard device error codes; it is set by the system.
- io_Actual. This is the actual number of bytes transferred by the Clipboard device internal routines during command execution. It usually specifies the number of bytes transferred between a task-defined buffer and the current clipboard file.
- io_Length. This is the number of bytes to be transferred between the current clipboard file and a task-defined buffer for a CMD_READ or CMD_WRITE command.
- io_Data. This points to the first byte of the task-defined buffer used in a CMD_READ or CMD_WRITE command. For a CBD_POST command, io_Data points to a MsgPort structure representing the task's satisfy message port.
- io_Offset. This is the byte-offset number for the current clipboard file or task-defined buffer. It is usually 0 for the first CMD_READ or CMD_WRITE

command in a sequential set of commands; the Clipboard device internal routines automatically update it for later commands.

- io_ClipID. This is the ordinal clip identifier for a CMD_READ or CMD_WRITE command. The Clipboard device internal routines specify this parameter for the first CMD_READ or CMD_WRITE command and maintain it for later sequential commands.

## The SatisfyMsg Structure

The SatisfyMsg structure is defined as follows:

```
struct SatisfyMsg {
   struct Message sm_Msg;
   UWORD sm_Unit;
   LONG sm_ClipID;
};
```

Its parameters have these meanings:

- sm_Msg. This is the name of a Message substructure used by the Clipboard device internal routines to send a message to a task, asking it to satisfy a previously dispatched CBD_POST command.

- sm_Unit. This is the number of the Clipboard device unit to which the CBD-_POST command was originally directed. This unit sends the satisfy message to a task's satisfy message port. The primary unit number for the Clipboard device is 0.

- sm_ClipID. This is the ordinal clip identifier assigned by the Clipboard device internal routines to the original CBD_POST command. The assigned value of this parameter maintains the association of the original CBD_POST command and the satisfy message.

## USE OF FUNCTIONS

## *CloseDevice*

### Syntax of Function Call

```
CloseDevice (iOClipReq)
             A1
```

## Purpose of Function

CloseDevice closes access to a unit of the Clipboard device. The IOClipReq structure io_Device and io_Unit parameters will be set to −1, indicating that the IOClipReq structure cannot be used again until these parameters are reinitialized by OpenDevice. CloseDevice also decrements the Device structure lib_OpenCnt parameter and the Unit structure unit_OpenCnt parameter by 1 for the specified unit. When these parameters are reduced to 0 for all units, a deferred expunge for the Clipboard device can take place.

When CloseDevice returns, a task cannot use the Clipboard device until it executes another OpenDevice function call. However, the current settings of the Clipboard device internal parameters are saved for the next call to OpenDevice by this or another task.

## Inputs to Function

| | |
|---|---|
| **iOClipReq** | A pointer to an IOClipReq structure; also a pointer to an IOStdReq structure |

## Discussion

CloseDevice terminates access to the device routines for a specified Clipboard device unit. If a number of Clipboard device units are open at the same time, the device will not be entirely closed until the last unit is closed.

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

A task can close the Clipboard device with a call to CloseDevice when it has finished its Clipboard device operations. Then another task that wants to use the device can open, write, read, and close it; the sequence can be repeated again and again in a C language program that uses Clipboard device routines. If multiple tasks want to pass data among themselves using the same Clipboard device unit, they must all have the unit open simultaneously.

## *OpenDevice*

## Syntax of Function Call

```
error = OpenDevice ("clipboard.device", unitNumber, iOClipReq, 0)
D0                 A0                  D0          A1         D1
```

## Purpose of Function

This function opens access to a specified unit of the Clipboard device and initializes certain parameters in the unit's IOClipReq structure to their most recently specified or default values. It also increments the Device structure lib_OpenCnt parameter and the Unit structure unit_OpenCnt parameter, thereby preventing a deferred expunge of the Clipboard device. Clipboard device units are always opened in shared access mode, the default. This allows multiple tasks to share data through the same clipboard file.

OpenDevice requires a properly initialized task reply port with a task signal bit allocated to it if the calling task wants to be signaled. The results of function execution are found in the following parameters:

- io_Device. This points to the Device structure that manages the unit once it is opened; it contains the information necessary to manage the unit.
- io_Unit. This points to the Unit structure that defines and manages a MsgPort structure for the unit; the MsgPort structure represents the device-unit request queue.
- io_Error. A 0 value here indicates that the request was successful. IOERR_OPENFAIL indicates that the unit could not be opened, most often due to lack of memory.

## Inputs to Function

| | |
|---|---|
| **"clipboard.device"** | A pointer to a null-terminated string representing the name of the Clipboard device |
| **unitNumber** | The Clipboard device unit number; use 0 if available |
| **iOClipReq** | A pointer to an IOClipReq structure |
| **0** | Indicates that the flags argument is not used |

## Preparation of the IOClipReq Structure

Initialize mn_ReplyPort to point to a MsgPort structure representing the task reply port. Set io_Command to 0, or set it to CMD_READ or CMD_WRITE if the task should open the Clipboard device and then perform a read or write operation immediately. Set all other IOStdReq substructure parameters to 0, or copy them from the IOClipReq structure of a previous OpenDevice call.

If the CreateExtIO function is used to create the IOClipReq structure (see Chapter 2), it must typecast its returned pointer value into a pointer to an IOClipReq structure. The pointer will then also point to the IOStdReq structure, which is the first entry in the IOClipReq structure.

### Discussion

OpenDevice is usually called to open the Clipboard device and to initialize parameters for a CMD_READ or CMD_WRITE command. Any parameters that are not explicitly initialized will retain their previous values or be set to the default values assigned by the Clipboard device routines. If the calling task wants to use other values for these parameters, it should initialize them after OpenDevice returns. When a task finishes its Clipboard device writing and reading, it should (but need not) close the device.

## STANDARD DEVICE COMMANDS

## *CMD_READ*

### Purpose of Command

CMD_READ causes a stream of characters to be read into a task-defined buffer from the current clipboard file. The number of characters is specified by the IOClipReq structure io_Length parameter, the clipboard-file offset location is specified by io_Offset, and the task buffer location is specified by io_Data.

Because CMD_READ does not support QuickIO, it always replies to the task reply-port queue. The results of command execution are found in io_Actual, io_ClipID, and io_Error. The number of characters actually read from the current clipboard file by CMD_READ are returned in io_Actual. The value of the io_ClipID parameter is specified by the Clipboard device internal routines; this value should be used for all subsequent sequential CMD_READ commands. A 0 value in io_Error indicates that the command was successful. IOERR_NOCMD indicates that the io_Command parameter was specified incorrectly; IOERR_BADLENGTH indicates that the io_Length parameter was specified incorrectly.

### Preparation of the IOClipReq 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 Unit structures that manage a specified Clipboard device unit; these can be copied from the IOClipReq structure initialized by the OpenDevice function call. Set io_Command to CMD_READ and io_Flags to 0. Also initialize the following parameters:

- io_Length. Set this to the number of characters to be read into the task-defined buffer from the current clipboard file.

- io_Data. Set this to point to the first byte in the task's read buffer. If you set this to 0, the Clipboard device internal routines will increment the io_Offset parameter by the value of io_Length, as if io_Length bytes had been read by CMD_READ. This is how a task performs a set of sequential reads for the current clipboard file.
- io_Offset. For a single CMD_READ operation, initialize this to the byte number within the current clipboard file. For sequential access of separate areas of the clipboard file from its beginning, initialize io_Offset to 0 for the first CMD_READ command dispatched by a task. Then do not change it for subsequent CMD_READ commands; the Clipboard device internal routines will update the io_Offset parameter so that the IOClipReq structure can be used for the next CMD_READ command in the sequence. When io_Offset points to a byte number beyond the end of the current clipboard file, CMD_READ signals the Clipboard device internal routines that the task has finished reading the file.
- io_ClipID. Initialize this to 0 for the first CMD_READ command dispatched by the task. The Clipboard device internal routines will then initialize this value in the reply.

## Discussion

As you saw in Figure 12.2, the current clipboard file can be read in sequential fashion. If the specified io_Offset parameter represents a byte number small enough to be within the current clipboard file, the task buffer is filled with data coming from the current clipboard file. The task should initialize the IOClipReq structure io_ClipID parameter to 0 for the first CMD_READ command; the Clipboard device internal routines will then specify this parameter internally, and it can be used for subsequent CMD_READ commands.

If a task initializes the IOClipReq structure io_Data parameter to 0, the Clipboard device internal routines will increment the IOClipReq structure io_Offset parameter by the value of io_Length. A task can also specify a large IOClipReq structure io_Length parameter so that the Clipboard device internal routines will move the byte pointer to the end of the current clipboard file.

Whenever a task is in the process of reading the current clipboard file, any attempts to write data into that file will be postponed until all pending CMD_READ commands have finished processing and have been replied to the task reply-port queue. Therefore, a task can use an io_Offset parameter that points beyond the end of the current clipboard file to signal the Clipboard device internal routines that all of that task's CMD_READ commands are finished. The task can then, once again, start writing data from a task-defined buffer to the current clipboard file.

## CMD_RESET

### Purpose of Command

CMD_RESET resets the specified unit to the boot-up time state as if it were just initialized. All Clipboard device internal parameters are set to their default values, but the io_Device and io_Unit parameters initialized by the original OpenDevice call remain valid.

CMD_RESET allows QuickIO and always replies to the task reply-port queue if the IOF_QUICK bit is not set. 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.

### Preparation of the IOClipReq 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 Unit structures that manage unit 0 of the Clipboard device; these can be copied from the IOClipReq structure initialized by the OpenDevice function call. Also set io_Command to CMD_RESET, and set io_Flags to 0 or to IOF_QUICK if QuickIO is desired.

### Discussion

The CMD_RESET command is quite simple in its actions: it merely reinitializes the Clipboard device internal parameters to their default (boot-up) values.

## CMD_UPDATE

### Purpose of Command

CMD_UPDATE informs a Clipboard device unit that all previously dispatched CMD_WRITE commands, including a sequential series, have been executed and replied. This lets the internal routines know that any pending CMD_READ requests can be dispatched and will be successful. CMD_UPDATE cannot be dispatched while any

CMD_WRITE commands are queued in the device-unit request queue or have not yet replied to the task reply-port queue.

Because it does not support QuickIO, CMD_UPDATE is always replied to the task reply-port queue. A 0 in io_Error indicates that the command was successful; IOERR-_NOCMD indicates that the io_Command parameter was specified incorrectly.

### Preparation of the IOClipReq 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 Unit structures that manage the specified Clipboard device unit; these can be copied from the IOClipReq structure initialized by the OpenDevice function call. Also set io_Command to CMD_UPDATE, and set io_Flags to 0.

### Discussion

A task dispatches CMD_UPDATE after it has dispatched one or more CMD_WRITE commands and has also verified that those requests have been replied to the dispatching task's reply-port queue. You can use the GetMsg, Remove, CheckIO, and WaitIO functions to determine that all of the CMD_WRITE commands have been processed and replied.

## *CMD_WRITE*

### Purpose of Command

CMD_WRITE causes a stream of characters to be written from a task-defined buffer into the current clipboard file. The number of characters is specified by the IOClipReq structure io_Length parameter, and the current buffer offset is specified by the IOClipReq structure io_Offset parameter. The first buffer byte is specified by io_Data.

A new io_ClipID parameter is obtained by clearing the io_ClipID parameter for the first CMD_WRITE command. Subsequent CMD_WRITE commands must not the alter this parameter. However, if a task dispatches a CMD_WRITE command in response to a signal from a replied SatisfyMsg structure for a pending CMD_POST command, the CMD_POST I/O request structure io_ClipID parameter must be copied for the CMD_WRITE command.

CMD_WRITE does not support QuickIO and always replies to the task reply-port queue. The results of command execution are found in the following parameters:

- io_Actual. This is the number of characters actually written from the task-defined buffer into the current clipboard file.
- io_ClipID. This value is specified automatically by the Clipboard device internal routines; it should be used for all subsequent sequential CMD_WRITE commands.
- io_Error. A 0 value here indicates that the command was successful. IO-ERR_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 IOClipReq 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 Unit structures that manage the Clipboard device unit; these can be copied from the IOClipReq structure initialized by the OpenDevice function call. Set io_Command to CMD_WRITE, and set io_Flags to 0. Also initialize the following parameters:

- io_Length. Set this to the number of characters to be written from the the task-defined write buffer to the current clipboard file.
- io_Data. Set this to point to the first byte in the task's write buffer. For sequential writes, the Clipboard device internal routines automatically increment this buffer pointer by io_Actual after each write.
- io_Offset. Set this to 0 for a single write or the initial write of a set of sequential writes. For subsequent writes, the Clipboard device internal routines automatically update this parameter to the next byte position. When io_Offset is specified as so large that it points to a byte number beyond the end of the current task-defined buffer, the clipboard file is automatically padded with zeros.
- io_ClipID. Initialize this to 0 for the first CMD_WRITE dispatched in a sequential series. The Clipboard device internal routines will then automatically initialize this value in the replied IOClipReq structure, and the value can be copied and used for subsequent CMD_WRITE commands. Copy this parameter from the io_ClipID of the I/O request structure representing a CMD_POST command if this write operation is being done to satisfy the CMD_POST command.

## Discussion

The CMD_WRITE command can perform either single write operations or sequential write operations. If the specified io_Offset parameter represents a byte number small enough to be within the specified task-defined buffer, CMD_WRITE acts as a single write operation; the current clipboard file is filled with data coming from the task-defined buffer.

For a set of sequential writes, the task should initialize the IOClipReq structure io_ClipID parameter to 0 for the first CMD_WRITE command; the Clipboard device internal routines will then initialize the io_ClipID parameter internally, and that value will be used for subsequent CMD_WRITE commands.

## DEVICE-SPECIFIC COMMANDS

## *CBD_CLIPREADID*

### Purpose of Command

The CBD_CLIPREADID command allows a task to determine the read clip identifier of the current CMD_READ command. If the current read clip identifier is greater than a CBD_POST command's ordinal clip identifier, the post data held privately by a task should not be pasted into the current clipboard file with CMD_WRITE.

CBD_CLIPREADID does not support QuickIO and always replies to the task reply-port queue. When CBD_CLIPREADID executes, the Clipboard device internal routines initialize its IOClipReq structure io_ClipID parameter to match that of the current CMD_READ command. A 0 value in io_Error indicates the command was successful; IOERR_NOCMD indicates that the io_Command parameter was specified incorrectly.

### Preparation of the IOClipReq 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 Unit structures that manage a specified Clipboard device unit; these can be copied from the IOClipReq structure initialized by the OpenDevice function call. Set io_Command to CBD_CLIPREADID, and initialize io_Flags, io_Length, io_Data, io_Offset, and io_ClipID to 0.

### Discussion

The CBD_CLIPREADID command allows a task to determine the current CMD_READ command's clip identifier. This identifier can be compared to that of a CBD_POST command, and the task can then decide if the deferred CMD_WRITE command associated with the CBD_POST command should be dispatched. If the CMD_READ clip identifier is greater than the CBD_POST clip identifier, the post data held by the task is too old to be pasted into the current clipboard file, and the task should proceed with other activities. If the CMD_READ clip identifier is equal to or less than the CBD_POST clip identifier, the post data is still current and the task should dispatch a CMD_WRITE command to write the data into the current clipboard file.

## *CBD_CLIPWRITEID*

### Purpose of Command

The CBD_CLIPWRITEID command allows a task to determine the write clip identifier of the current CBD_WRITE command. This identifier can be compared to the ordinal clip identifier of a CBD_POST command. If the write clip identifier is greater than the CBD_POST identifier, then the post data held privately by a task is obsolete; the CBD-_POST command no longer needs to be satisfied.

Because CBD_CLIPWRITEID does not support QuickIO, it always replies to the task reply-port queue. When CBD_CLIPWRITEID executes, the Clipboard device internal routines will initialize its io_ClipID parameter to that of the current CMD_WRITE command. A 0 value in io_Error indicates the command was successful; IOERR-_NOCMD indicates that the io_Command parameter was specified incorrectly.

### Preparation of the IOClipReq 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 Unit structures that manage a specified Clipboard device unit; these can be copied from the IOClipReq structure initialized by the OpenDevice function call. Set io_Command to CBD-_CLIPWRITEID, and initialize io_Flags, io_Length, io_Data, io_Offset, and io_ClipID to 0.

### Discussion

The CBD_CLIPWRITEID command allows a task to determine the current CMD_WRITE io_ClipID parameter; this identifier can then be compared to the clip identifier of a CBD-_POST command. The comparison allows a task to decide if the deferred write should be dispatched. If the CMD_WRITE clip identifier is greater than the CBD_POST clip identifier, the post data held by the task is too old to be pasted into the current clipboard file, and the task should proceed with other activities. If the CMD_WRITE clip identifier is equal to or less than the CBD_POST clip identifier, the post data is still current and the task should dispatch the CMD_WRITE command.

## CBD_POST

### Purpose of Command

The CBD_POST command allows a task to post a clip to the current clipboard file. It tells the Clipboard device internal routines to postpone a CMD_WRITE command until they determine that a task needs the data contained in the command. In this way, CBD-_POST acts as a deferred CMD_WRITE command.

Because CBD_POST does not support QuickIO, it always replies to the task reply-port queue. When the command executes, the io_ClipID parameter contains the value assigned to CBD_POST by the Clipboard device internal routines. A 0 in io_Error indicates that the task was successful; IOERR_NOCMD indicates that the io_Command parameter was specified incorrectly.

### Preparation of the IOClipReq 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 Unit structures that manage a specified Clipboard device unit; these can be copied from the IOClipReq structure initialized by the OpenDevice function call. Set io_Command to CMD_POST, and initialize io_Flags, io_Length, io_Offset, and io_ClipID to 0. Set io_Data to point to a MsgPort structure representing a satisfy message port; use the CreatePort function to create this port and to assign a signal bit number to it. This port can belong to a task other than the one dispatching the CBD_POST command.

### Discussion

CBD_POST allows a task to delay the dispatching of a CMD_WRITE command until the data in the command is needed by a clipboard file that is managed by that task or another task. In this way, one task can send clipboard data bytes to the current clipboard file of another task, which means one task can defer a CMD_WRITE command for another task. CBD_POST should only be used if a clipboard data clip is in a task-private data format, is large, or is changing frequently. You should not convert it to an IFF (Interchange File Format) form.

When the Clipboard device internal routines determine that a CBD_POST command must be satisfied (in other words, a task needs the clipboard information), they send a satisfy message and signal to the task, which must then dispatch a CMD_WRITE command immediately in order to satisfy the original needs of the CBD_POST command. The SatisfyMsg structure representing the satisfy message should then be removed from the task's satisfy message port so that it will be empty when the next message arrives.

If a task needs to determine if a CBD_POST command represents the most current clipboard clip, it should compare the replied CBD_POST io_ClipID parameter with the io_ClipID parameter returned by a newly dispatched CBD_CURRENTREADID command. If the parameter returned by CBD_CURRENTREADID is greater than that of CBD_POST, the clip is no longer current. If a task has a pending CBD_POST command and needs to determine if it should satisfy it (before the task exits, for example), it should compare the CBD_POST and a newly dispatched CBD_CURRENTWRITEID io_ClipID parameter. If the parameter returned by CBD_CURRENTWRITEID is greater than that of CBD_POST, the task does not need to satisfy the CBD_POST command.

# 13 The Timer Device

## Introduction

The Timer device provides a timing mechanism for your tasks. It is ROM-resident and is loaded into WCS memory from the Kickstart disk when the Amiga is first booted. The Timer device is opened automatically by AmigaDOS and the Parallel, Serial, Console, and Input devices; it can be opened by an OpenDevice function call as well.

The Timer device signals a task when a specified amount of time has elapsed. However, because the Amiga is a multitasking system, the Timer device cannot always guarantee that the *exact* amount of time has elapsed; the actual amount will always be greater than or equal to the time specified in the request.

You can program the Timer device with OpenDevice, CloseDevice, and three device-specific functions: AddTime, CmpTime, and SubTime. These device-specific functions allow a task to perform time arithmetic, which can increase the accuracy of the Timer device as it processes time-interval requests. In addition, three device-specific commands—TR_ADDREQUEST, TR_SETSYSTIME, and TR_GETSYSTIME—allow a task to establish time intervals and to ascertain or change the system time.

## Operation of the Timer Device

Figure 13.1 illustrates the general operation of the Timer device, which has two units. Unit 0, referred to as UNIT_MICROHZ, is dedicated to high-resolution timing jobs. Unit 1, referred to as UNIT_VBLANK, is dedicated to low-resolution timing jobs. Both units can be opened only in shared access mode.

Requests are dispatched to the Timer device internal routines with the DoIO or SendIO function; the BeginIO function is not usually used to access the Timer device. A task should use DoIO when it needs to dispatch a single request, which will be processed when it gets to the top of the device-unit request queue. In the meantime, the requesting task will sleep while it waits for a reply; it will be signaled of the reply's arrival in its reply port when the Timer device has processed it.

A task should use SendIO when it needs to dispatch multiple I/O requests to the Timer device. The requests will be queued in the device-unit request queue with requests already there. The Timer device internal routines will reply to the task reply-port queue when they have finished with each request, and they will then signal the task. In the meantime, the task could have retained execution and progressed to another point in its task statement sequence.

A task can create as many reply ports as it needs. If your task had three high-resolution timing jobs, for example, it could create three reply-port queues by declaring and initializing three TimeRequest structures for TR_ADDREQUEST commands and specifying different mn_ReplyPort parameters in each. As usual, the task can use GetMsg or Remove to remove reply messages from the queues when commands are dispatched with SendIO.

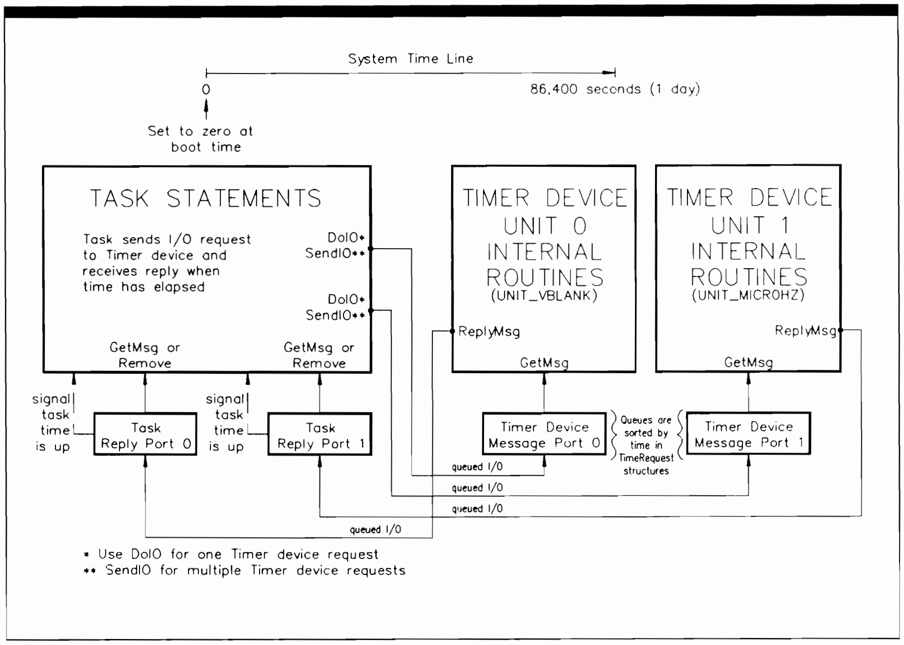

System Time Line
0
86,400 seconds (1 day)
Set to zero at boot time
TASK STATEMENTS
Task sends I/O request to Timer device and receives reply when time has elapsed
DoIO*
SendIO**
DoIO*
SendIO**
GetMsg or Remove
GetMsg or Remove
signal task time is up
Task Reply Port 0
signal task time is up
Task Reply Port 1
TIMER DEVICE UNIT 0 INTERNAL ROUTINES (UNIT_VBLANK)
ReplyMsg
GetMsg
TIMER DEVICE UNIT 1 INTERNAL ROUTINES (UNIT_MICROHZ)
ReplyMsg
GetMsg
Timer Device Message Port 0
Queues are sorted by time in TimeRequest structures
Timer Device Message Port 1
queued I/O
queued I/O
queued I/O
queued I/O
* Use DoIO for one Timer device request
** SendIO for multiple Timer device requests

**Figure 13.1:** *Operation of the Timer Device*

The Timer device request queue operates differently from most device request queues. TR_ADDREQUEST commands are sorted according to the time specified in their TimeRequest structures. This ensures that the earliest TR_ADDREQUEST request is placed at the top of the queue. Sorting is performed continuously by the Timer device internal routines.

In addition to the timing signals in the TR_ADDREQUEST command, the Timer device works with the system time through the TR_GETSYSTIME and TR_SETSYSTIME commands. System time ranges from 0 to 86,400 seconds. The AddTime, CmpTime, and SubTime functions allow the Timer device to add, subtract, and compare system times for a task that needs this information.

## Timer Units and Timer Arithmetic

As you saw in Figure 13.1, the Timer device has two units: UNIT_MICROHZ and UNIT_VBLANK. UNIT_MICROHZ uses a programmable timer in the 8250 CIA (complex interface adapter) chip. This chip is really two distinct chips: 8250 CIA A and B. The A chip registers are located in RAM at addresses BFE001-BFEF01; the B chip registers are at addresses BFD000-BFDF00. Four register addresses (BFE401-BFE701) in the A chip are devoted to timing calculations, as are four register addresses in the B chip (BFD400-BFD700). See the software memory map in the *ROM Kernel Manual* for more about these chips and their registers.

UNIT_MICROHZ has a timing resolution of one microsecond, which means that it increments its internal counters every microsecond. However, this unit tends to get behind when the system is busy—it has a great deal of system overhead to deal with. For this reason, it is not very accurate over long periods of time, especially if the machine is busy with a number of tasks.

UNIT_VBLANK uses a vertical-blanking interrupt as a signal to increment its internal timing counter. On the American versions of the Amiga, the counter is incremented 60 times per second, which means that it is accurate to within ±16.67 milliseconds. The specified time interval and the actual time interval can differ by this amount regardless of how busy the machine is with its other tasks.

The characteristics of the two Timer device units determine how a task should specify a time interval. For example, if a task needed a 20-second time interval, it could dispatch a TR_ADDREQUEST command to the UNIT_VBLANK unit specifying the tv_Secs parameter as 20 and tv_Micros as 0. UNIT_VBLANK would then determine how many whole 1/60-of-a-second intervals were in the total 20 seconds and signal the task when that number of intervals had elapsed; the task would actually be signaled at 20 seconds ±16.67 milliseconds. If the task needed a 1.5-second interval, it could again use the UNIT_VBLANK unit, specifying a tv_Micros of 1.0 and a tv_Secs of 500,000. The task would not be signaled at the precise 1.5-second interval, because 1.5 is not evenly divisible by ±16.67 milliseconds.

However, a task is not required to use UNIT_VBLANK when the time interval exceeds one second—it can use UNIT_MICROHZ. To do so, it must specify the tv_Micros parameter as a value between 0 and 1,000,000. Therefore, if a task needs 1.5-second intervals, it can specify tv_Secs as 1.0 and tv_Micros as 500,000, and then dispatch the TR_ADDREQUEST command to UNIT_MICROHZ. The resulting intervals could be more accurate than those produced by UNIT_VBLANK—unless the system was extremely busy. As you can see, a task often needs to perform time arithmetic in order to determine the best method for establishing time intervals.

## Time-Interval Corrections in a Busy System

The AddTime, CmpTime, and SubTime functions together with the TR_SETSYSTIME and TR_GETSYSTIME commands allow a task to make a series of timing intervals more accurate. The task can recalibrate its calculations at intermediate points in the time-interval progression. This is best illustrated with an example showing how TR_ADDREQUEST, TR_GETSYSTIME, and SubTime can achieve greater accuracy in time-interval calculations.

Suppose a task needed to be signaled every second for a one-minute period. It could use UNIT_MICROHZ and send a series of TR_ADDREQUEST commands to that unit. Each of these commands would set tv_Secs to 0 and tv_Micros to 1,000,000.

If the system was only dealing with that task, the UNIT_MICROHZ counter would be devoted to counting down the specified time period. Each time the tv_Micros parameter reached 1,000,000, a TR_ADDREQUEST request would be replied to the task reply-port queue, and the task would be signaled. When the system is dedicated to one task, the calculated time intervals are indeed accurate.

On the other hand, if the system was working with a number of system and programmer tasks and trying to satisfy the timing needs of all of those tasks simultaneously, UNIT_MICROHZ would get behind in incrementing the tv_Micros parameter representing the current TR_ADDREQUEST command. Although the first signal might arrive precisely at one second, the second and subsequent signals would tend to arrive later than the one-second interval. This timing error would increase as more timing signals were sent to the task.

A task cannot preclude these types of timing errors—it must adjust for their presence. It can do so in the following way:

1. The task should dispatch the first TR_ADDREQUEST command representing the one-second signal. It should then dispatch a QuickIO TR_GETSYSTIME command immediately and save the returned value in a task-local variable (tv_Secs = Time1) in the replied TR_GETSYSTIME command TimeRequest structure.
2. The task should then program a loop to dispatch a series of 59 additional TR_ADDREQUEST commands to UNIT_MICROHZ with DoIO so that it sleeps until they have all replied. Again, the task should dispatch a QuickIO TR_GETSYSTIME command immediately and save the value returned in a second task-local variable (tv_Secs = Time2).
3. The task should then use the SubTime function to subtract Time1 from Time2 and compare the difference to 60. If the system is extremely busy, the difference in system times will be greater than 60 seconds. The busier the system, the greater this difference will be. As an example, call the difference (Time2 – Time1) DeltaT and form a ratio: DeltaT/60. If the system is dedicated entirely to the task, DeltaT will be 0 (60 – 60) and the ratio will also be 0, meaning that calculated time intervals are indeed accurate. If the system is so busy that DeltaT is 60 (120 – 60), the actual elapsed time will be two minutes, when one minute was intended.
4. Assuming that the system will remain equally busy when the task dispatches these 60 TR_ADDREQUEST commands again, the task should specify the TimeRequest structure TimeVal substructure tv_Micros parameter as 500,000 rather than as 1,000,000. Then the newly dispatched TR_ADDREQUEST commands will indeed signal the task at one-second time intervals.

This type of recalibration can be carried further, with the task computing the ratio corrector on a finer time scale. For this example, a second TR_GETSYSTIME command could be dispatched after the fifth TR_ADDREQUEST command, and the elapsed system time indicated by the TR_GETSYSTIME could be compared to 5.0 seconds. A time-interval ratio could again be calculated to correct the tv_Micros parameter for subsequent TR_ADDREQUEST commands. The same thing could be done after the tenth TR_ADDREQUEST command. In fact, a TR_GETSYSTIME command could be dispatched after every TR_ADDREQUEST command. However, with many TR_GETSYSTIME and TR_ADDREQUEST commands, the time used for function and command execution plays an important role in the accuracy of these procedures; you must investigate the tradeoffs involved in the circumstances of each timing loop and plan a strategy accordingly.

## Timer Device Commands

The Timer device does not work with standard device commands. Instead, it has three device-specific commands—TR_ADDREQUEST, TR_GETSYSTIME, and TR_SETSYSTIME. All three commands support queued I/O as well as QuickIO. None of them support automatic immediate-mode operation. All three commands affect the TimeRequest structure io_Error parameter; TR_GETSYSTIME also affects the TimeVal values.

### Sending Commands to the Timer Device

Figure 13.2 depicts the general scheme used to dispatch commands to the Timer device internal routines. The lines with arrows represent the parameters you should initialize and those returned by the Timer device internal routines.

The programming process for the Timer device consists of three phases:

1. TimeRequest structure preparation. You have complete control over this phase—here, you initialize parameters in the TimeRequest structure in preparation for dispatching a command to the Timer device internal routines. The parameters include the usual ones required by most devices, as well as the TimeVal structure tv_Secs and tv_Micros parameters. In all cases, the choice of parameters to initialize depends on the command you plan to dispatch. These parameters provide an information path to the data needed by the Timer device internal routines in order to process the command.

2. Timer device processing. The only part you play in this phase is to dispatch the command to the device using the DoIO or SendIO function. Control then passes to the device and system internal routines.

3. Command output parameter processing. The system and Timer device internal routines have complete control over this phase. Here, the results of Timer device command processing have been returned to the task that issued the command. If it was not a successful QuickIO request, it was processed when it moved to the top of the device-unit request queue; the I/O request is in the task reply-port queue awaiting

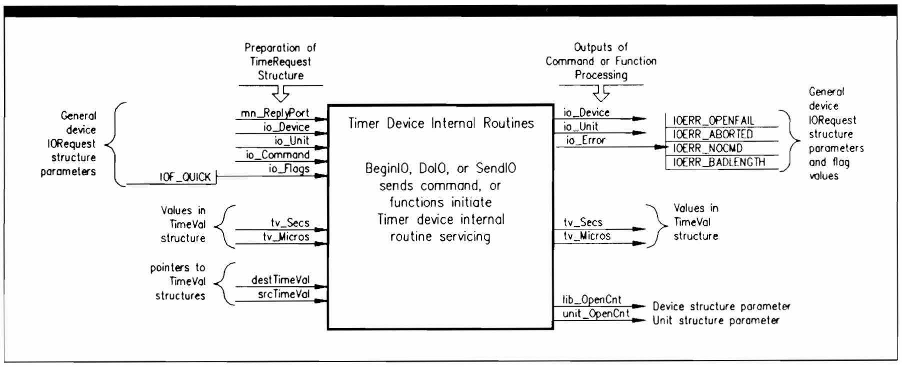

Preparation of TimeRequest Structure
Outputs of Command or Function Processing
General device IORequest structure parameters
mn_ReplyPort
io_Device
io_Unit
io_Command
io_Flags
IOF_QUICK
Values in TimeVal structure
tv_Secs
tv_Micros
pointers to TimeVal structures
destTimeVal
srcTimeVal
Timer Device Internal Routines
BeginIO, DoIO, or SendIO sends command, or functions initiate Timer device internal routine servicing
io_Device
io_Unit
io_Error
IOERR_OPENFAIL
IOERR_ABORTED
IOERR_NOCMD
IOERR_BADLENGTH
General device IORequest structure parameters and flag values
tv_Secs
tv_Micros
Values in TimeVal structure
lib_OpenCnt
unit_OpenCnt
Device structure parameter
Unit structure parameter

**Figure 13.2:** *Timer Device Command and Function Processing*

the task's processing. If the request was a successful QuickIO request, it came back directly to the requesting task. The returned parameters direct you to data appropriate for your task.

Two Timer device commands (TR_ADDREQUEST and TR_GETSYSTIME) provide output parameters; however, the tv_Secs and tv_Micros values returned by TR_ADDREQUEST are meaningless. All three Timer device commands provide the io_Error parameter as output.

Figure 13.2 also depicts the parameters that play a part in Timer device function setup and processing. The Timer device has three device-specific functions: AddTime, SubTime, and CmpTime; all three work with pointers to the TimeVal structure. The OpenDevice and CloseDevice functions affect the Unit structure unit_OpenCnt parameter and the Device structure lib_OpenCnt parameter; OpenDevice also affects the io_Error parameter.

## Structures for the Timer Device

The Timer device works with two structures, TimeRequest and TimeVal, as shown in Figure 13.3. The TimeRequest structure is used to formulate I/O requests to be dispatched to the Timer device internal routines. The TimeVal structure represents a system time for the TR_GETSYSTIME and TR_SETSYSTIME commands: it represents a time interval for the TR_ADDREQUEST command and the AddTime, SubTime, and CmpTime functions.

### The TimeVal Structure

The TimeVal structure is defined as follows:

```
struct TimeVal {
   ULONG tv_Secs;
   ULONG tv_Micros;
};
```

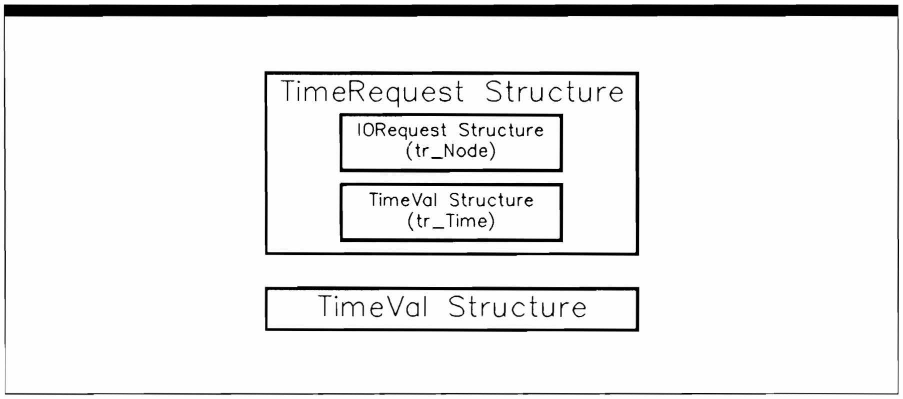


**Figure 13.3:** *Timer Device Structures*

Its parameters have the following meanings:

- tv_Secs is the value in seconds of the time request.
- tv_Micros is the value in microseconds of the time request. This value is between 0 and 1,000,000.

### The TimeRequest Structure

The TimeRequest structure is defined as follows:

```
struct TimeRequest {
   struct IORequest tr_Node;
   struct timeval tr_Time;
};
```

Its parameters have the following meanings:

- tr_Node is name of an IORequest substructure used to represent Timer device I/O requests for the TR_ADDREQUEST, TR_GETSYSTIME, and TR_SETSYSTIME commands. Its io_Message, io_Device, io_Unit, io_Command, io_Flags, and io_Error parameters are used in the usual way. For the AddTime, CmpTime, and SubTime functions, the io_Device parameter is used with the TimerBase parameter to gain entrance into the Timer device routine library and to locate the vector offsets of routines representing these three functions.
- tr_Time is the name of the TimeVal substructure representing the requested time interval for the TR_ADDREQUEST command and the system time for the TR_SETSYSTIME and TR_GETSYSTIME commands.

## USE OF FUNCTIONS

## *AddTime*

### Syntax of Function Call

```
AddTime (destTimeVal, srcTimeVal)
         A0           A1
```

### Purpose of Function

This function adds the tv_Micros and tv_Secs parameters of one TimeVal structure to another. The result is stored in the second TimeVal structure's tv_Micros and tv_Secs parameters.

To access this function, a task must initialize a system-global variable named Timerbase to point to the Device structure, which manages the Timer device unit. It does this by opening the Timer device unit with OpenDevice and assigning Timerbase to the TimeRequest io_Device parameter returned by OpenDevice, as shown here:

```
APTR TimerBase;
Timerbase = (APTR)TimeRequest->In_Node.io_Device);
```

The Timer device internal routines will then be able to locate the vector offset for the device library routines corresponding to the AddTime function. The A0 and A1 pointer register values will be unchanged when AddTime returns.

## Inputs to Function

| | |
|---|---|
| **srcTimeVal** | A pointer to a TimeVal structure |
| **destTimeVal** | A pointer to a second TimeVal structure |

## Discussion

The AddTime function allows a task to add the time values represented by two TimeVal structures and to perform time arithmetic in order to compensate for the limited resolution of UNIT_VBLANK, the limited accuracy of the UNIT_MICROHZ, and the inaccuracy of system time values returned by the TR_GETSYSTIME command when the system is busy.

Being a function, AddTime makes direct entries into the Timer device routine library. Therefore, a task does not need to prepare a TimeRequest structure to access it. However, the task must open the Timer device with a call to OpenDevice if it is not already opened. It must then initialize the Timerbase variable to point to the Device structure returned in the TimeRequest structure io_Device parameter of the OpenDevice call. Because the Device structure acts as an interface for the Timer device internal routines and commands, for these functions, it provides the path to the specific function vector offsets in the device library as well.

## *CloseDevice*

### Syntax of Function Call

```
CloseDevice (timeRequest)
                 A1
```

### Purpose of Function

This function closes access to a Timer device unit. When CloseDevice returns, the current task cannot use the unit until it executes another OpenDevice function call. However, the current settings of the Timer device internal parameters are saved for the next OpenDevice function call by this or other tasks.

CloseDevice decrements the lib_OpenCnt parameter in the Device structure and the unit_OpenCnt parameter in the Unit structure for the specified unit, reducing each of these by 1. Once these parameters are reduced to 0, a deferred expunge of the Timer device can take place.

### Inputs to Function

| | |
|---|---|
| **timeRequest** | A pointer to a TimeRequest structure; also a pointer to an IORequest structure |

### Discussion

A task should always verify that the Timer device internal routines have replied all of its requests before it calls the CloseDevice function to close that unit. It can do so by using the GetMsg, Remove, CheckIO, and WaitIO functions to see what requests are in the task reply-port queue.

The Timer device operates in shared access mode; therefore, when a task is done with its Timer device unit operations, it can (but need not) close the unit with a call to CloseDevice.

## *CmpTime*

### Syntax of Function Call

```
result = CmpTime (firstTimeVal, secondTimeVal)
                  A0            A1
```

### Purpose of Function

This function compares the tv_Micros and tv_Secs parameters of one TimeVal structure to those in another. To use CmpTime, a task must initialize a system global variable named Timerbase to point to the Device structure that manages the specified Timer device unit. See the AddTime function discussion to learn how this is done.

The results of function execution are expressed as integer values:

- A 0 value indicates that the tv_Micros and tv_Secs parameters of the first and second TimeVal structures are equal.
- A +1 indicates that the tv_Micros and tv_Secs parameters of the first TimeVal structure represent less time than those of the second TimeVal structure.
- A −1 indicates that the tv_Micros and tv_Secs parameters of the first TimeVal structure represent more time than those of the second TimeVal structure.

### Inputs to Function

| | |
|---|---|
| **firstTimeVal** | A pointer to a TimeVal structure |
| **secondTimeVal** | A pointer to a second TimeVal structure |

### Discussion

The CmpTime function allows a task to compare two times by comparing the TimeVal structure tv_Secs and tv_Micros parameters that represent those times. The task can then take appropriate action based on these comparisons. See the discussion of the AddTime function to learn more about the time arithmetic involved.

## *OpenDevice*

### Syntax of Function Call

```
error = OpenDevice ("timer.device", unitNumber, timeRequest, 0)
D0                   A0              D0          A1          D1
```

### Purpose of Function

This function opens access to the internal routines of a specified unit of the Timer device. Once OpenDevice has successfully opened a Timer device unit, it then initializes certain parameters in the TimeRequest structure. OpenDevice also increments the Device structure lib_OpenCnt parameter and the Unit structure unit_OpenCnt parameter, thereby preventing a deferred expunge of the Timer device.

OpenDevice requires a properly initialized task reply port with a task signal bit allocated to it if the calling task needs to be signaled when commands are replied. The results of function execution are as follows:

- io_Device. This points to a Device structure that manages the Timer device unit once it has been opened. The Device structure contains all the information necessary to manage the unit and to reach the data and routines in the Timer device library for the unit.
- io_Unit. This points to a Unit structure that will be used to define and manage a MsgPort structure for a Timer device unit; the MsgPort structure represents the device-unit request queue.
- io_Error. A 0 value indicates that the requested open succeeded. IOERR_OPENFAIL indicates that the specified unit could not be opened. This could be caused by a lack of memory or by an attempt to open the same unit in a task twice without first closing it.

## Inputs to Function

| | |
|---|---|
| **"timer.device"** | A pointer to a null-terminated string representing the name of the Timer device |
| **unitNumber** | The Timer device unit number; use either UNIT_VBLANK or UNIT_MICROHZ for this argument |
| **timeRequest** | A pointer to a TimeRequest structure |
| **0** | Indicates that the flags argument is not used |

## Preparation of the TimeRequest Structure

Initialize mn_ReplyPort to point to a MsgPort structure for the task reply port that will receive the request replies when the Timer device internal routines have finished processing OpenDevice and other Timer device commands. Initialize io_Command to 0 or to a Timer device command if the task should open the device and dispatch a command immediately.

Because Timer device I/O requests automatically destroy TimeVal structure parameters (tv_Secs and tv_Micros) before they are replied, these parameters must always be reinitialized even if they have the same values as a previous TimeVal structure.

If the CreateExtIO function is used to allocate and initialize the TimeRequest structure (see Chapter 2), it should typecast its returned pointer value (to an IORequest structure) into a pointer to a TimeRequest structure. The pointer would then also point to the IORequest structure that is the first entry in the TimeRequest structure.

### Discussion

Keep in mind that the Timer device is opened automatically by AmigaDOS and by the Parallel, Serial, Console, and Input devices. If a task has already opened these devices or is working in AmigaDOS, it will not have to open the Timer device explicitly. In fact, such an OpenDevice call will probably result in a failed attempt to open an already opened Timer device unit, thereby returning an IOERR_OPENFAIL error.

Once a task opens a Timer device unit, it can dispatch a series of commands to work with the system time and to provide timing events. When a task has finished dispatching its Timer device commands, it should close the unit with a call to CloseDevice.

## *SubTime*

### Syntax of Function Call

```
SubTime (destTimeVal, srcTimeVal)
         A0           A1
```

### Purpose of Function

This function subtracts the tv_Micros and tv_Secs parameters of one TimeVal structure from those of another. The result is stored in the tv_Micros and tv_Secs parameters of the second TimeVal structure. Because this function is in the set of Timer device internal routines, a task must initialize a system-global variable named Timerbase to point to the Device structure that manages a specified Timer device unit. See the AddTime function discussion to learn how this is done.

### Inputs to Function

| | |
|---|---|
| **srcTimeVal** | A pointer to a TimeVal structure |
| **destTimeVal** | A pointer to a second TimeVal structure |

### Discussion

The SubTime function allows a task to subtract the time values represented by two TimeVal structures. See the discussion of the AddTime function to learn more about the time arithmetic involved.

## DEVICE-SPECIFIC COMMANDS

## TR_ADDREQUEST

### Purpose of Command

TR_ADDREQUEST allows a task to time its operations. It directs the Timer device internal routines to count down the time interval specified in the command's TimeRequest structure. The Timer device internal routines sort and incorporate the command's request with other Timer device I/O requests in the system; they reply the request structure to the task reply-port queue and signal the reply task when they count down the specified time interval to zero.

TR_ADDREQUEST allows QuickIO and always replies to the task reply-port queue if QuickIO is unsuccessful. A 0 value in io_Error indicates that the command was successful. IOERR_NOCMD indicates that the io_Command parameter was specified incorrectly.

The TimeRequest structure tv_Secs and tv_Micros parameters do not contain any meaningful values once the command executes. Therefore, if you want to use the TimeRequest structure for another TR_ADDREQUEST command, you must reinitialize them before dispatching the new command. The task reply port should be defined with a signal bit number so that the dispatching task will know when the TR_ADDREQUEST structure is replied; it can then go on with the activities it should perform when this time interval has elapsed.

### Preparation of the TimeRequest 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 Unit structures that manage the specified Timer device unit; these can be copied from the TimeRequest structure initialized by the OpenDevice function call. Set io_Command to TR_ADDREQUEST. Initialize io_Flags to IOF_QUICK for QuickIO; otherwise, set it to 0. Initialize tv_Secs to the number of seconds contained in the desired time interval, and initialize tv_Micros to the number of microseconds it contains.

### Discussion

When a task uses the SendIO function to dispatch a series of TR_ADDREQUEST commands, the Timer device internal routines signal the task when the specified time interval has elapsed. The task can then use the CheckIO, WaitIO, and GetMsg functions to deal with the replies in its reply-port queue. As soon as the task detects a reply —when the time interval represented by the TimeRequest structure has elapsed—it can take appropriate action. If a task uses the DoIO function to dispatch a single TR_ADDREQUEST request, it will sleep until the specified time interval has elapsed.

The order in which TR_ADDREQUEST commands are dispatched is not important; the Timer device internal routines will sort the TimeRequest structures in the device-unit request queue according to the specified time intervals represented in the structures and will process them accordingly.

TR_ADDREQUEST can be dispatched as a QuickIO request. With QuickIO, the Timer device internal routines will not signal the dispatching task. Therefore, the best use for QuickIO is when the task needs a specific time delay but does not need to be signaled. In this case it is best to dispatch TR_ADDREQUEST with the DoIO function; the task will be held back as needed and will then go on to the next program statement when the time delay has elapsed. You can also use the Exec Delay function for this purpose (see Volume I).

## TR_GETSYSTIME

### Purpose of Command

This command allows a task to ascertain the current system time. System time is initialized by the system as 0 seconds at boot-up time and runs to 86,400 seconds (the number of seconds in one day), at which point the system reinitializes it to 0. The system time value is incremented by one time unit whenever a video vertical-blanking event occurs. System time units are incremented in vertical-blanking intervals of 1/60 of a second. System time is also incremented automatically by one unit each time a task asks for the system time using the TR_GETSYSTIME command. Because of this internal procedure, the system time value returned by TR_GETSYSTIME is always unique and unrepeating.

TR_GETSYSTIME allows QuickIO and always replies to the task reply-port queue if QuickIO is not unsuccessful. The results of command execution are found in tv_Secs, which indicates the number of seconds for the current system time; and in tv_Micros, which indicates the number of microseconds. The io_Error parameter returns a 0 value if the command was successful; IOERR_NOCMD indicates that the io_Command parameter was specified incorrectly.

### Preparation of the TimeRequest 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 Unit structures that manage a specified Timer device unit; these can always be copied from the

TimeRequest structure initialized by the OpenDevice function call. Set io_Command to TR_GETSYSTIME. Initialize io_Flags to IOF_QUICK for QuickIO, and set tv_Secs and tv_Micros to 0.

### Discussion

The TR_GETSYSTIME and TR_SETSYSTIME commands should be dispatched as QuickIO commands. Without QuickIO, the requests would be queued in the device-unit request queue and replied to the task reply-port queue. Processing of the queue would take time, which would make the system time returned by these two commands invalid.

## *TR_SETSYSTIME*

### Purpose of Command

This command allows a task to set the current system time. System time is initialized automatically to 0 seconds at boot-up and increases to 86,400 seconds (the number of seconds in one day), at which point the system reinitializes it to 0. If a task needs to reinitialize the system time back to 0, it should use the TR_SETSYSTIME command to reset the values in the TimeRequest structure TimeVal substructure tv_Micros and tv_Secs parameters. The system time value is incremented one time unit whenever a video vertical-blanking event occurs; the units are incremented by vertical-blanking intervals of 1/60 of a second.

TR_SETSYSTIME allows QuickIO and always replies to the task reply-port queue if QuickIO is unsuccessful. 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.

### Preparation of the TimeRequest 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 Unit structures that manage the specified Timer device unit; these can be copied from the TimeRequest structure initialized by the OpenDevice function call. Set io_Command to TR_SETSYSTIME. Initialize io_Flags to IOF_QUICK for QuickIO. Initialize tv_Secs to the number of seconds you want for the system time (this must be between 0 and 86,400 seconds—one day), and set tv_Micros to the number of microseconds.

## Discussion

The TR_SETSYSTIME command allows a task to set the system time forward or backward as necessary to satisfy the current timing needs of the task. Like TR_GETSYSTIME, it should always be dispatched as a QuickIO request.

# The TrackDisk Device

14

## Introduction

The TrackDisk device controls the Amiga disk drives and motors. It also writes and reads raw and encoded data to and from the disk tracks, provides disk-system status information to a task, and allows you to add a series of disk-change interrupt routines to the system. The TrackDisk device is the lowest-level software interface to the Amiga disk system, and it is used by AmigaDOS. It is a ROM-resident device and is loaded automatically into ROM when the WCS ROM is written from the Kickstart disk at boot-up time.

There are some standard device commands that the TrackDisk device does not support, but device-specific equivalents are provided. This device is programmed with the CloseDevice and OpenDevice functions and three structures: IOExtTD, TDU_PublicUnit, and BootBlock.

## Operation of the TrackDisk Device

Figure 14.1 illustrates the general operation of the TrackDisk device, which supports both 3.5-inch and 5.25-inch floppy disks. It also supports hard disks.

The TrackDisk device has four units, which can only be opened in exclusive access mode. Unit 0 is dedicated to the Amiga internal disk drive; units 1, 2, and 3 are dedicated to the first, second, and third external disk drives. The external disk drives can be

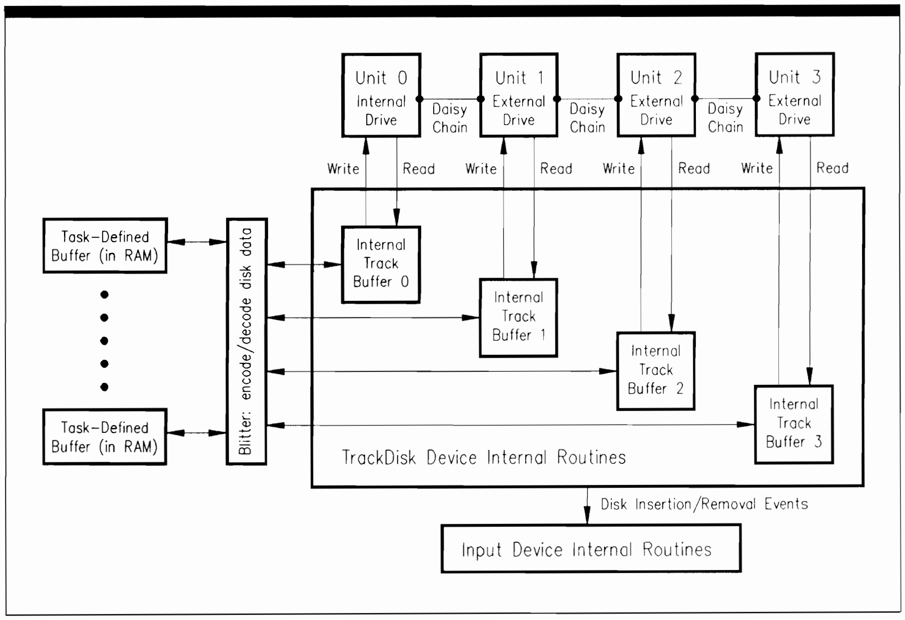


**Figure 14.1:** *Operation of the TrackDisk Device*

daisy-chained, as shown in the figure. The first connector is at the back of the Amiga. Table 14.1 summarizes the pin connections.

The TrackDisk device is a block-oriented device; data is read in and out in full track blocks. Because QuickIO is intended for single-character operations, it would not speed system response and is therefore not provided for this device. Instead, I/O requests are placed in a device-unit request queue.

Each unit also has an internal track buffer that contains data for all 11 sectors in a track. Track buffers are managed by the TrackDisk device internal routines; a task has only limited control over them. They act as intermediate data locations for raw or preprocessed data sent between the task-defined buffers and the disk media. All task-defined data-transfer buffers must be in the first 512K of memory.

Read operations (the TD_RAWREAD, ETD_READ, and ETD_RAWREAD commands) read one track of data from a disk into the track buffer and then send the data to the task-defined buffer; for the ETD_READ command, the data is decoded into an internally compatible form by the Blitter coprocessor during the transfer. Write operations (the TD_RAWWRITE, ETD_RAWWRITE, and ETD_WRITE commands) write one or more sectors of data from a task-defined buffer to the track buffer and onto the disk; for

**Table 14.1:** *Pin Connections for External Disk Drives*

| Pin No | Data at Pin | Data Direction | Description |
|---|---|---|---|
| 1 | RDY | Input/output | Disk installed or identification mode |
| 2 | DKRD | Input | MFM (modified frequency modulation) input data to Amiga |
| 3–7 | GND | | Ground pins |
| 8 | MTRXD | Open collector | Motor on data |
| 9 | SEL2B | Open collector | Select drive 2 |
| 10 | DRESB | Open collector | Amiga system reset |
| 11 | CHNG | Input/output | Toggle for disk installed/not installed |
| 12 | +5V | | 270ma maximum; 410ma surge |
| 13 | SIDEB | Output | Side 1 if active, side 0 if inactive |
| 14 | WPRO | Input/output | Write-protected disk |
| 15 | TKO | Input/output | Read–write head over track 0 |
| 16 | DKWDB | Open collector | Write gate (enable) to drive |
| 17 | DKWDB | Open collector | MFM output data from Amiga |
| 18 | STEPB | Open collector | Selected drive steps one drive in DIRB direction |
| 19 | DIRB | Open collector | Head step direction |
| 20 | SEL3B | Open collector | Select drive 3 |
| 21 | SEL1B | Open collector | Select drive 1 |
| 22 | INDEX | Input/output | Index pulse, once per revolution |
| 23 | +12V | | 160ma maximum; 540ma surge |

the ETD_WRITE command, data is encoded into an externally compatible form by the Blitter coprocessor.

The TrackDisk device also sends disk insertion and removal events to the Input device if it has been opened. A task can use the TD_REMOVE command to execute a software interrupt routine when a disk is inserted or removed. See Chapter 7 for more information on the the Input device and its interactions with the TrackDisk device.

## TrackDisk Device/Floppy-Disk Interactions

Figure 14.2 illustrates TrackDisk device operations on a floppy disk. Each 3.5-inch floppy disk has two sides, 80 tracks to a side, 11 sectors to a track, and 512 data bytes to a sector. Track numbers increase from 0 for the outside track to 79 for the inside track. In addition, each sector of a disk can contain 16 bytes of sector-identification information. The 3.5-inch floppy disk can hold a total of 880K of information (440K to a side). Other types of floppy disks have their own characteristics.

Several TrackDisk device operations affect information on the disk in the internal track buffers or in the task-defined buffers—read operations, write operations, update operations, and formatting operations:

- Each read or write operation can transfer one or more sectors between the floppy disk and the task-defined buffer; the number of sectors is controlled by the task statements that define the data transfer. If a task executes a command to read sector data from a disk (unit) and that data is already in the unit's internal track buffer, no immediate disk activity will occur. If the sector data is not in the internal track buffer, however, a full track will be read into it automatically. If the internal track

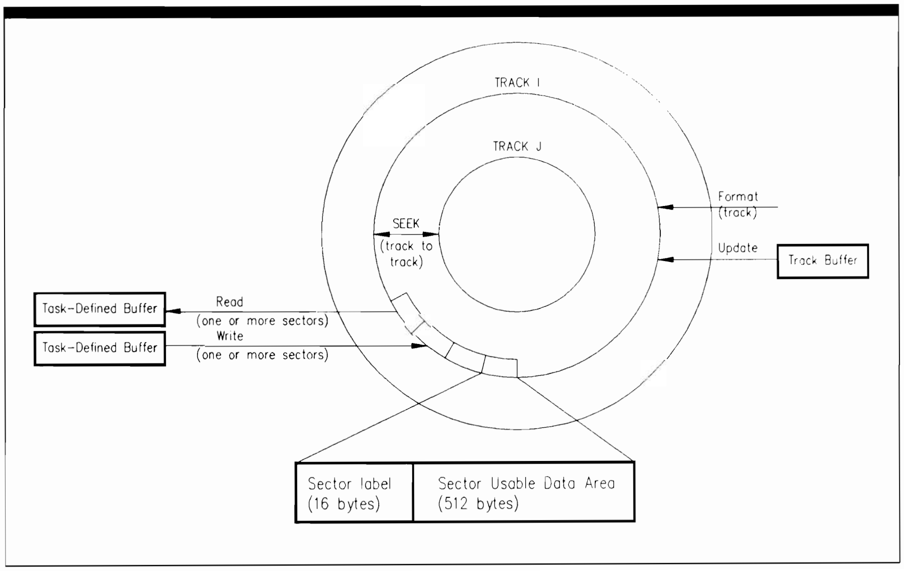


**Figure 14.2:** *Interactions of a Floppy Disk and the TrackDisk Device*

buffer currently contains data that has not been written to the disk since that data was last changed, the older data will be written to the disk first. In the same way, no immediate disk activity will occur for a write operation unless that operation would write over data already in the track buffer.

Reading from and writing to the disk occurs only if the current contents of the internal track buffer would not be lost by the execution of the read or write operation. The system always knows if sector data is currently in the track buffer but not on the disk; it takes appropriate action to protect task-defined data at all times. Because the device internal track buffer contains all 11 sectors of a track, this automatic bookkeeping operation minimizes disk activity.

- Update operations (ETD_CLEAR and ETD_UPDATE) allow you to write the contents of all track buffers to the floppy or hard disk in an emergency situation, such as a power failure.
- Formatting operations (TD_FORMAT and ETD_FORMAT) for a floppy disk are done on a track-by-track basis, which speeds the formatting process considerably.

In addition, as the figure shows, the TrackDisk device TD_SEEK command causes the floppy disk read–write head to move from track to track.

## TrackDisk Device Commands

The TrackDisk device does not work directly with standard device commands. Instead, it uses device-specific commands, which fall into two categories: "normal" commands and extended commands. Extended commands (beginning with ETD) perform some of the same operations as normal (TD) commands, except that they also deal with disk-change or sector-label information. The extended commands were added to prevent inadvertent read or write operations when a disk is changed.

### Sending Commands to the TrackDisk Device

Figures 14.3(a) and (b) depict the general scheme used to send commands to the TrackDisk device routines. The lines with arrows in Figure 14.3(a) represent the parameters you should initialize, and those in Figure 14.3(b) represent the parameters returned by the TrackDisk device internal routines.

The TrackDisk device programming process consists of three phases:

1. Structure preparation. You have complete control over this phase; here, you initialize parameters in the IOExtTD structure in preparation for sending a command or function to the TrackDisk device internal routines. The parameters include those required by most devices; in addition, the iotd_SecLabel and iotd_Count parameters are used for the extended commands. Taken together, these parameters provide an information path to the data needed by the TrackDisk device in order to process the command.

2. TrackDisk device processing. The only part you play in this phase is to send the command to the device using the BeginIO, DoIO, or SendIO function. Control then passes to the device and system internal routines.

3. Command output parameter processing. The system and TrackDisk device routines have complete control over the values found in this phase; here, the results of command processing have been returned to the task that originally issued the command. The I/O request was processed when it moved to the top of the TrackDisk device request queue, and it is now in the task reply-port queue awaiting the task's processing.

As Figure 14.3(b) shows, TD_GETDRIVETYPE and TD_GETNUMTRACKS provide the io_Actual parameter as output, and all commands provide the io_Error parameter.

Figures 14.3(a) and (b) also depict the parameters that play a part in TrackDisk device function setup and processing. The OpenDevice and CloseDevice functions affect the Unit structure unit_OpenCnt parameter and the Device structure lib_OpenCnt parameter; OpenDevice also affects the io_Error parameter.

## Structures for the TrackDisk Device

The TrackDisk device works with three device-specific structures: IOExtTD, TDU_Public-Unit, and BootBlock. All TrackDisk device commands are represented by the IOExtTD

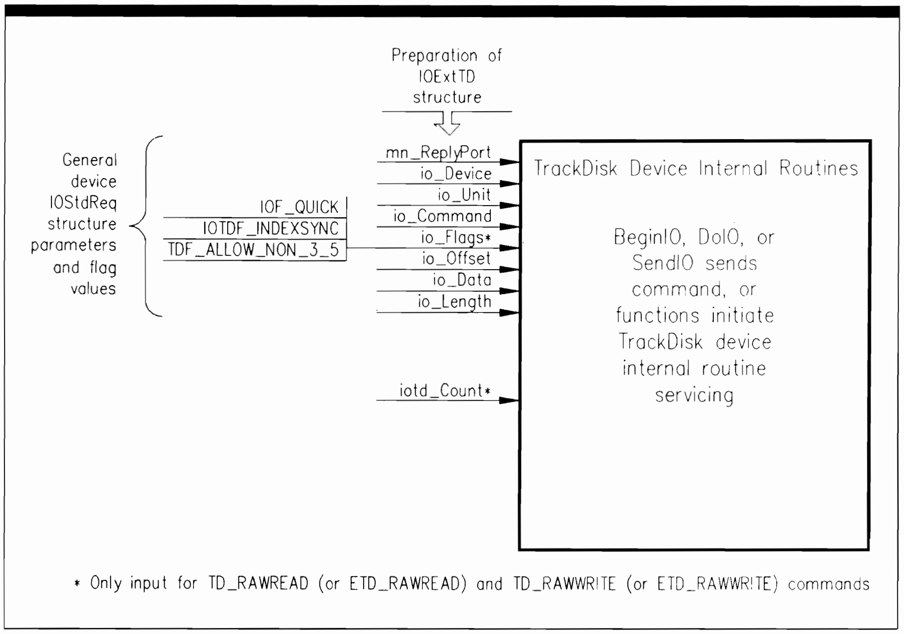

Preparation of IOExtTD structure
General device IOStdReq structure parameters and flag values
IOF_QUICK
IOTDF_INDEXSYNC
TDF_ALLOW_NON_3_5
mn_ReplyPort
io_Device
io_Unit
io_Command
io_Flags*
io_Offset
io_Data
io_Length
iotd_Count*
TrackDisk Device Internal Routines
BeginIO, DoIO, or SendIO sends command, or functions initiate TrackDisk device internal routine servicing
* Only input for TD_RAWREAD (or ETD_RAWREAD) and TD_RAWWRITE (or ETD_RAWWRITE) commands

**Figure 14.3(a):** *TrackDisk Device Command and Function Processing (Specifications)*

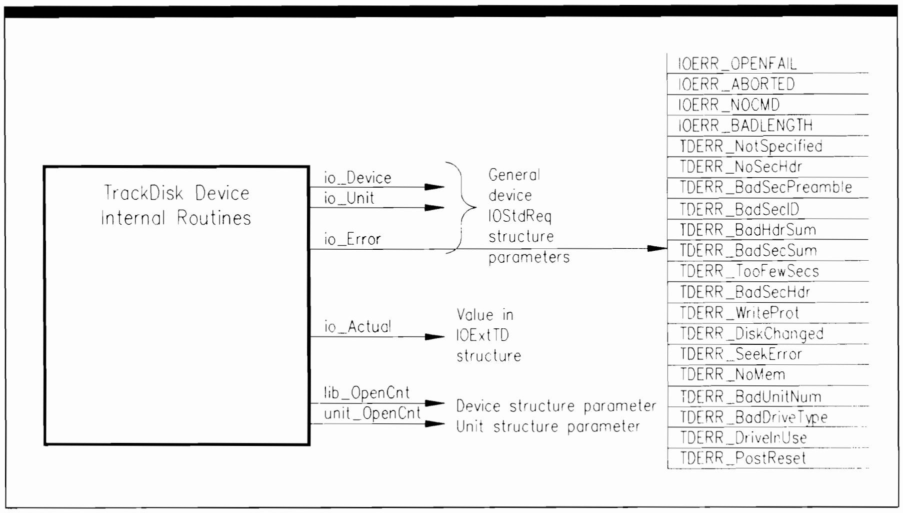


**Figure 14.3(b):** *TrackDisk Device Command and Function Processing (Outputs)*

structure. The TDU_PublicUnit structure is used to manage a device-unit request queue and to control timing and other aspects of disk operation. The BootBlock structure is used to define the two-sector (1K) bootblock for each disk that the Amiga recognizes. These structures are shown in Figure 14.4.

## The IOExtTD Structure

The IOExtTD structure is defined as follows:

```
struct IOExtTD {
   struct IOStdReq iotd_Req;
   ULONG iotd_Count;
   ULONG iotd_SecLabel;
};
```

Its parameters have the following meanings:

- iotd_Req. This is the name of an IOStdReq substructure that represents the message-passing, error-reporting, and task-defined buffer specifications for a command.
- iotd_Count. This is the minimum value allowed for the disk-change counter variable, which keeps track of the number of disk insertions and removals. The current value of the disk-change counter variable in an I/O request is returned by a TD_CHANGENUM command. If the current value is less than this structure's iotd_Count parameter, the internal device routines will fail to process the I/O request and reply it with the IOExtTD structure io_Error parameter set to TDERR_DiskChanged. The task should then request the user to insert the appropriate disk. When the correct disk is inserted, the TrackDisk device internal routines will recognize its label; another TD_CHANGENUM will return a

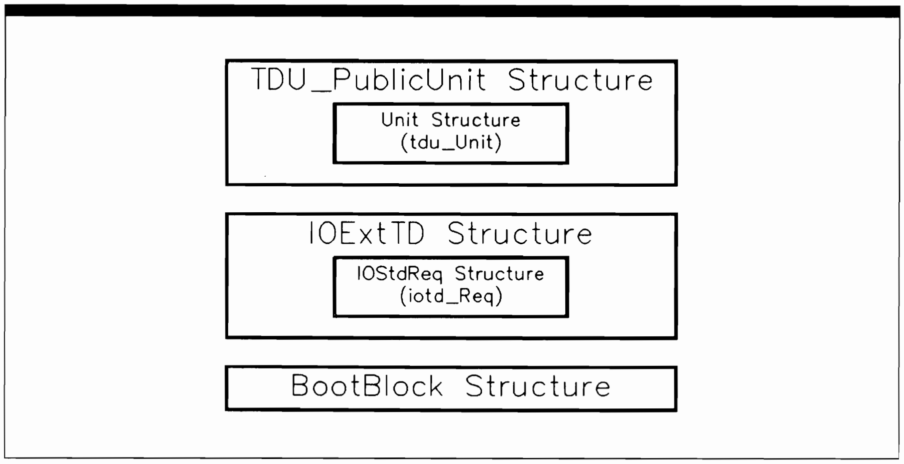


**Figure 14.4:** *TrackDisk Device Structures*

disk-change counter value that is consistent with the value of the IOExtTD structure iotd_Count parameter, and the I/O request can be sent again.

- iotd_SecLabel. This points to a series of contiguous 16-byte buffers containing sector-label information for read and write operations. The system will read or write this identification information to or from the disks.

## The TDU_PublicUnit Structure

The TDU_PublicUnit structure is defined as follows:

```
struct TDU_PublicUnit {
   struct Unit tdu_Unit;
   UWORD tdu_Comp01Track;
   UWORD tdu_Comp10Track;
   UWORD tdu_Comp11Track;
   ULONG tdu_StepDelay;
   ULONG tdu_SettleDelay;
   UBYTE tdu_RetryCnt;
};
```

Its parameters have the following meanings:

- tdu_Unit. This is the name of a Unit substructure used to manage a TrackDisk device unit. It contains a MsgPort substructure to manage the device-unit message port, and it also contains the unit_OpenCnt parameter, which keeps track of the number of tasks using a unit. Because the TrackDisk device operates in exclusive access mode, Unit_OpenCnt will always have a value of either 0 or 1.

A Unit structure's parameters are always system-private—only the system can write to them. However, TDU_PublicUnit has been extended so that a task can examine and modify the following public parameters:

- tdu_Comp01Track, tdu_Comp10Track, and tdu_Comp11Track. These are the track numbers to use for the first, second, and third track precompensations.
- tdu_StepDelay. This is the time (in milliseconds) to wait after track-to-track stepping the disk motor. As of Release 1.2, the step delay is 3.6 milliseconds. However, this parameter is hardware-dependent; a value that works with a 68000 CPU may not work with a 68010 or 68020 CPU, which have faster clock rates.
- tdu_SettleDelay. This is the time (in milliseconds) to wait after a track-to-track seek operation. This parameter is also hardware-dependent.
- tdu_RetryCnt. This is the number of times the seek operation should be reattempted. A forced seek can be accomplished using the TD_Seek or ETD_Seek command; however, this should only be done for diagnostic purposes.

## The BootBlock Structure

The BootBlock structure is defined as follows:

```
struct BootBlock {
   UBYTE bb_id[4];
   LONG bb_chksum;
   LONG bb_dosblock;
};
```

Its parameters have the following meanings:

- bb_id[4]. This is a four-character identifier that defines the type of disk in the disk drive. At the present time, it can be initialized to "DOS0," indicating a valid DOS disk, or "KICK," indicating the Kickstart disk.
- bb_chksum. This is the current value of the checksum for the boot block. It is used to verify the integrity of the disk when booting.
- bb_dosblock. This parameter is reserved for future DOS use.

## TrackDisk Device Error Codes

Error codes are returned by the TrackDisk device functions and commands in the IOExtTD structure io_Error parameter. They have the following meanings:

- A 0 value always indicates that the command or function was successful.

- IOERR_OPENFAIL, IOERR_ABORTED, IOERR_NOCMD, and IOERR-_BADLENGTH have the same meanings for the TrackDisk device as they do for other devices (see Chapter 3).
- TDERR_NotSpecified. This is a "catch-all" error code. It usually means that the error is not one of the following errors.
- TDERR_NoSecHdr. The TrackDisk device could not find the sector header for a sector.
- TDERR_BadSecPreamble. The TrackDisk device could not find the sector preamble for a sector.
- TDERR_BadSecID. A sector has bad sector-label information.
- TDERR_BadHdrSum. A sector has a bad sector-header checksum.
- TDERR_BadSecSum. A sector has a bad sector-data checksum.
- TDERR_TooFewSecs. The TrackDisk device cannot find enough sectors to satisfy the requirements of the command.
- TDERR_BadSecHdr. A sector has bad sector-header data.
- TDERR_WriteProt. A command tried to write to a protected disk.
- TDERR_DiskChanged. There is no disk in the drive, or the disk-change counter value and the IOExtTD structure iotd_Count parameter are not consistent.
- TDERR_SeekError. The seek operation could not find sector 0 on a disk.
- TDERR_NoMem. The system ran out of memory trying to perform the function or command.
- TDERR_BadUnitNum. A function or command asked for a unit number inconsistent with the number of units currently in the system (units 0, 1, 2, and 3). The system also returns this error if a command or function addresses a unit that does not have a disk driver connected to it.
- TDERR_BadDriveType. The drive type specified in a function or command is not one the system understands.
- TDERR_DriveInUse. The drive is being used by another task. The current task must wait until the task now using the unit closes it.
- TDERR_PostReset. The user has posted a reset of the system by using the Ctrl/left-Amiga/right-Amiga key combination.

## USE OF FUNCTIONS

## CloseDevice

### Syntax of Function Call

```
CloseDevice (iOExtTD)
             A1
```

### Purpose of Function

This function closes access to a specific unit of the TrackDisk device. When CloseDevice returns, the IOExtTD structure io_Device and io_Unit parameters will be set to −1, and no task can use the IOExtTD structure until it is reinitialized by an OpenDevice call. CloseDevice also decrements the Device structure lib_OpenCnt parameter and the Unit structure unit_OpenCnt parameter to indicate that the task no longer has the unit open. If unit_OpenCnt is reduced to 0 for all open units of the TrackDisk device, a deferred expunge can take place.

### Inputs to Function

| | |
|---|---|
| **iOExtTD** | A pointer to an IOExtTD structure |

### Discussion

Each time a task opens a TrackDisk device unit, it must execute a CloseDevice call for the unit before another task can open it. When a task opens a unit, the Unit structure unit_OpenCnt parameter is incremented to indicate that the unit is presently owned by the task; no other task can open it until unit_OpenCnt is once again reduced to 0.

## OpenDevice

### Syntax of Function Call

```
error = OpenDevice ("trackdisk.device", unitNumber, iOExtTD, flags)
D0                  A0                  D0          A1       D1
```

### Purpose of Function

This function opens access to a specific TrackDisk device unit. OpenDevice initializes the IOExtTD structure io_Device and io_Unit parameters to point to Device and Unit structures that the system uses to manage the unit. OpenDevice also increments the Device structure lib_OpenCnt and the Unit structure unit_OpenCnt parameters, thereby preventing a deferred expunge of the TrackDisk device.

### Inputs to Function

| | |
|---|---|
| **"trackdisk.device"** | A pointer to a null-terminated string representing the name of the TrackDisk device |
| **unitNumber** | The TrackDisk device unit number (0, 1, 2, or 3) |
| **iOExtTD** | A pointer to an IOExtTD structure |
| **flags** | Set this to 0 for 3.5-inch disks; set it to TDF_ALLOW_NON_3_5 for 5.25-inch disks |

### Preparation of the IOExtTD Structure

Initialize mn_ReplyPort to point to the MsgPort structure representing the task reply port.

### Discussion

If a unit is opened with OpenDevice, it must be closed with a call to CloseDevice before another task can use it—the TrackDisk device uses exclusive access mode only. Assign a message-port signal bit number if you want your task to be signaled when the OpenDevice I/O request is replied.

## DEVICE-SPECIFIC COMMANDS

## *ETD_CLEAR*

### Purpose of Command

This command marks the specified track internal buffer as invalid, forcing a reread of the disk on the next operation.

### Preparation of the IOExtTD 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 Unit structures that manage the addressed unit of the TrackDisk device; these parameters can be copied from the IOExtTD structure initialized by OpenDevice. Set io_Command to ETD-_CLEAR, and set io_Flags to 0.

### Discussion

The ETD_CLEAR command allows a task to mark an internal track buffer as invalid. This is useful if the task writes something into that buffer and later decides that the data is invalid. ETD_CLEAR does not affect the contents of task-defined buffers.

## ETD_RAWREAD

### Purpose of Command

This command reads raw data bits from a unit of the TrackDisk device. The system seeks a specific track and then reads one or more data sectors into a task-defined buffer. The system does not use the Blitter to decode the data. This command is identical to the TD_RAWREAD command, except that it is used when the task needs to verify the disk or work with sector-label information.

### Preparation of the IOExtTD 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 Unit structures that manage the addressed unit of the TrackDisk device; these parameters can be copied from the IOExtTD structure initialized by OpenDevice. Set io_Command to ETD_RAWREAD. Also set the following command-specific parameters:

- io_Flags. Set this parameter to 0, or set it to IOTDF_INDEXSYNC if you want the TrackDisk device to try to read the track data bits from the index mark on the track. It may or may not succeed; however, keep in mind that there will always be a delay in the reading process—perhaps a very long delay if, for example, interrupts have been disabled. See the TD_RAWREAD command for more information on these delays.

- io_Length. Set this to the length of the task-defined buffer in bytes; the maximum length is 32K.
- io_Data. Set this to point to the task-defined buffer to which the raw track data will be sent. This buffer must be in chip memory (MEMF_CHIP).
- io_Offset. Set this to the number of the track to be read into the task-defined buffer. For standard device commands (such as ETD_READ), io_Offset represents the number of bytes from the beginning of the disk; however, for ETD_RAW-READ, it represents the track number.
- iotd_Count. Set this to the maximum disk change-counter value.
- iotd_SecLabel. Set this parameter to 0, or set it to point to a series of contiguous-RAM 16-byte buffers—one buffer for each track sector to which sector-label information should be added during the read operation.

### Discussion

ETD_RAWREAD is used for raw data disk read operations in which the task must work with disk-change or sector-label information. No preprocessing of the track data will occur during a data transfer with this command; the data bits placed in the task-defined buffer will be exactly as they were on the disk. Because the data will be arranged in MFM (modified frequency modulation) format, you should use this command only if you know how MFM data is defined and used. In addition, if you use this command, your program may not be compatible with future releases of Amiga software. Your tasks can use the ETD_READ command to read data that has been decoded by the Blitter.

## *ETD_RAWWRITE*

### Purpose of Command

This command writes raw data bits to a unit of the TrackDisk device. It transfers the bits from a task-defined buffer to the disk without altering them through the action of the Blitter.

### Preparation of the IOExtTD 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 Unit structures that manage the addressed unit of the TrackDisk device; these parameters can be copied from the IOExtTD structure initialized by OpenDevice. Set io_Command to ETD_RAWWRITE. Also set the following command-specific parameters:

- io_Flags. Set this to 0, or set it to IOTDF_INDEXSYNC if you want the Track-Disk device to try to write the track data from the index mark on the track. It may or may not succeed; however, there will always be a delay—perhaps a very long delay if, for example, interrupts have been disabled.
- io_Length. Set this to the length in bytes of the task-defined buffer. The maximum length is 32K.
- io_Data. Set this to point to the task-defined buffer from which the raw track data bits will be copied. This buffer must be in chip memory (MEMF_CHIP).
- io_Offset. Set this to the track number to which the data bits should be written.
- iotd_Count. Set this to the maximum change-counter value.
- iotd_SecLabel. Set this to 0, or set it to point to a series of contiguous-RAM 16-byte buffers, one for each track sector to which you want to add sector-label information.

### Discussion

ETD_RAWWRITE is used for raw data write operations that require disk-change or sector-label information. No Blitter preprocessing of the data will occur during this data transfer; the data bits will be written to the disk exactly as they were in the task-defined buffer. Because they will be arranged in MFM format, you should use this command only if you know how MFM data is defined and used. In addition, if you use ETD-_RAWWRITE, your program may not be compatible with future releases of Amiga software. Your tasks can use the ETD_WRITE command to write data that has been encoded by the Blitter.

## ETD_UPDATE

### Purpose of Command

This command allows a task to write the current contents of a device-unit internal track buffer onto a disk connected to the unit. The contents of task-defined buffers are not affected by this operation.

### Preparation of the IOExtTD 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 Unit structures that manage the addressed unit of the TrackDisk device; these parameters can be copied from the IOExtTD structure initialized by OpenDevice. Also set io_Command to ETD_UPDATE, and set io_Flags to 0.

### Discussion

The ETD_UPDATE command allows a task to write the current contents of an internal track buffer onto a disk. Although this is done automatically in the event of a power failure or if the user initiates a machine reset, a task can clear the contents of the track buffer for other reasons. ETD_UPDATE writes out the internal track buffer only if that buffer has been changed since it was last read in. It does not affect contents of task-defined buffers.

## TD_ADDCHANGEINT

### Purpose of Command

This command adds a new disk-change interrupt mechanism to the TrackDisk device software system. It was implemented by the Amiga developers because the TD_REMOVE command was not fast enough to handle added interrupts correctly.

### Preparation of the IOExtTD 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 Unit structures that manage the addressed unit of the TrackDisk device; these parameters can be copied from the IOExtTD structure initialized by OpenDevice. Set io_Command to TD_ADD-CHANGEINT, and set io_Flags to 0. Also initialize io_Data to point to the Interrupt structure representing the disk-change interrupt you want to add to the system.

### Discussion

The TD_ADDCHANGEINT command allows the TrackDisk device system to support an extendable list of disk-change software interrupts, which allows disks to be inserted in or removed from a disk drive. You can design your own disk-change interrupts as you desire.

## *TD_CHANGENUM*

### Purpose of Command

This function returns the current value of the disk-change counter for a specified unit of the TrackDisk device. The value is returned in the IOExtTD structure io_Actual parameter.

### Preparation of the IOExtTD 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 Unit structures that manage the addressed unit of the TrackDisk device; these parameters can be copied from the IOExtTD structure initialized by OpenDevice. Also set io_Command to TD_CHANGENUM, and set io_Flags to 0.

### Discussion

The disk-change counter is important to the operation of the extended commands in the TrackDisk device software system. Each time a disk is inserted or removed in a particular unit, the disk-change counter for that unit is incremented. When the disk-change counter is greater than the value in the IOExtTD structure iotd_Count parameter, an extended command treats it as an error. This allows old I/O requests to be replied to the sending task after a disk has been changed and prevents the use of the wrong disk after a series of disk changes.

If a task needs to use extended commands but does not require disk removal information, the iotd_Count parameter should be set to the maximum unsigned long integer value, hexadecimal 0xFFFFFFFF.

## *TD_CHANGESTATE*

### Purpose of Command

This command determines if a disk is present in the drive of the specified unit. It returns a nonzero value in the IOExtTD structure io_Actual parameter if there is no disk in the drive.

### Preparation of the IOExtTD 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 Unit structures that manage the addressed unit of the TrackDisk device; these parameters can be copied from the IOExtTD structure initialized by OpenDevice. Also set io_Command to TD_CHANGESTATE, and set io_Flags to 0.

### Discussion

TD_CHANGESTATE allows a task to ascertain if a particular unit of the TrackDisk device has a disk in the disk drive. The task must know this before it goes on to read from or write to the disk.

## *TD_FORMAT*

### Purpose of Command

This command formats a disk, writing over any data already on it. It fills all sectors with the contents of the data specified in the IOExtTD structure. The IOExtTD structure io_Data parameter points to a task-defined buffer that holds the data-byte definitions to be used in the formatting process. It must point to at least one sector of formatting information. If the task-defined buffer contains more than 512 bytes of data, the first 512 bytes are used and the extra bytes are ignored. TD_FORMAT performs no error-checking.

### Preparation of the IOExtTD 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 Unit structures that manage the addressed unit of the TrackDisk device; these parameters can be copied from the IOExtTD structure initialized by OpenDevice. Also set io_Command to TD_FORMAT, and set io_Flags to 0. Set io_Data to point to at least one sector's worth of formatting bytes (at least 512 bytes). Data beyond one sector's worth will be ignored.

### Discussion

The TD_FORMAT command is used to format an entire disk with new formatting information. This is usually done to initialize a new disk, but it can also be done to reformat a damaged disk. The task should read each formatted track into a task-defined buffer to verify that the formatting is correct. A task can also use the ETD_FORMAT command if it wants to specify sector-label information and verify that the correct disk is in the drive.

## TD_GETDRIVETYPE

### Purpose of Command

This command returns the drive type to the calling task. Drive-type constants are small integers; they are defined in the Trackdisk.h and Trackdisk.i INCLUDE files. The drive type connected to the TrackDisk device unit is returned in the IORequest structure io_Actual parameter, where 1 indicates DRIVE3_5 (a standard 3.5-inch disk) or 2 indicates DRIVE5_25 (a 5.25-inch, IBM-type disk).

### Preparation of the IOExtTD 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 Unit structures that manage the addressed unit of the device; these parameters can be copied from the IOExtTD structure initialized by OpenDevice. Set io_Command to TD_GETDRIVETYPE, and set io_Flags to 0.

### Discussion

A task's OpenDevice call will fail if the device internal routines do not recognize a particular drive associated with a TrackDisk device unit. See the Disk.h INCLUDE file for a list of the raw drive identifiers recognized by the system.

## TD_GETNUMTRACKS

### Purpose of Command

With this command, the number of tracks available on a TrackDisk device unit is returned in the IOExtTD structure io_Actual parameter. The number is computed by the TrackDisk device internal routines based on the disk's physical characteristics.

### Preparation of the IOExtTD 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 Unit structures that manage the addressed unit of the TrackDisk device; these parameters can be copied from the IOExtTD structure initialized by OpenDevice. Set io_Command to TD_GET-NUMTRACKS and set io_Flags to 0.

### Discussion

The TD_GETNUMTRACKS command removes some restrictions that existed in Release 1.1 and earlier releases. In particular, this command makes the NUMTRACKS INCLUDE file constant found in Releases 1.0 and 1.1 obsolete.

## *TD_MOTOR*

### Purpose of Command

This command allows a task to turn the disk motor on and off. The disk motor is turned on automatically during the processing of most TrackDisk device I/O requests. However, it is not turned off automatically—you must use the TD_MOTOR or ETD_MOTOR command. The requested state of the disk motor (on or off) is specified by the IOExtTD structure io_Length parameter; a value of 1 will turn the motor on, and 0 will turn it off. The previous state of the motor is provided in the IOExtTD structure io_Actual parameter when TD_MOTOR is replied. A value of 1 means the motor was on, and 0 means it was off. A task can also use the ETD_MOTOR command if it wants to verify that the correct disk is in the drive before the motor is turned on or off.

### Preparation of the IOExtTD 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 Unit structures that manage the addressed unit of the TrackDisk device; these parameters can be copied from the IOExtTD structure initialized by OpenDevice. Also set io_Command to TD_MOTOR, and set io_Flags to 0.

### Discussion

Under most circumstances, turning the motor on is not necessary; each command sent to the TrackDisk device internal routines will do so. However, turning the motor off is the task's responsibility. Keep in mind that it is only safe to remove a disk if the motor is off.

## *TD_PROTSTATUS*

### Purpose of Command

This command allows a task to inquire about the current protection status of a disk. The protection status is returned in the IOExtTD structure io_Actual parameter, where a zero value indicates that the disk is unprotected and a nonzero value indicates that it is protected. If there is no disk in the drive, the IOExtTD structure io_Error parameter is set to TDERR_DiskChanged.

### Preparation of the IOExtTD 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 Unit structures that manage the addressed unit of the TrackDisk device; these parameters can be copied from the IOExtTD structure initialized by OpenDevice. Also set io_Command to TD_PROTSTATUS, and set io_Flags to 0.

### Discussion

TD_PROTSTATUS allows a task to determine if a particular unit of the TrackDisk device has a write-protected disk in its disk drive. The task must know this before it goes on to read from or write to the disk.

## *TD_RAWREAD*

### Purpose of Command

This command reads raw data bits from a unit of the TrackDisk device. It causes the system to seek a specific track and then read the track's data into a task-defined buffer. It is used for reading a disk when there is no need to verify that the disk has been changed or to work with sector-label information.

## Preparation of the IOExtTD 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 Unit structures that manage the addressed unit of the TrackDisk device; these parameters can be copied from the IOExtTD structure initialized by OpenDevice. Set io_Command TD_RAWREAD. Also set the following command-specific parameters:

- io_Flags. Set this to 0, or set it to IOTDF_INDEXSYNC if you want the Track-Disk device to try to read the track data bits from the index mark on the track. It may or may not succeed; however, there will always be a delay—perhaps a long delay if, for example, interrupts have been disabled. See the next section for more on IOTDF_INDEXSYNC.

- io_Length. Set this to the length (in bytes) of the task-defined buffer. The maximum length is 32K.

- io_Data. Set this to point to the task-defined buffer to which the raw track data will be sent. This buffer must be in chip memory (MEMF_CHIP).

- io_Offset. Set this to the track number to be read into the task-defined buffer. A normal device command (such as ETD_READ) always treats io_Offset as the number of bytes from the beginning of the disk. However, TD_RAWREAD treats it as the track number.

## Discussion

TD_RAWREAD is used for read operations in which no Blitter preprocessing of the track data should occur during the data transfer—the data bits placed in the task-defined buffer will be exactly as they were on the disk. Because the data will be arranged in MFM format, you should use this command only if you know how MFM data is defined and used. In addition, if you use TD_RAWREAD, your program may not be compatible with future releases of Amiga software.

TD_RAWREAD commands with IOTDF_INDEXSYNC set always contain a delay between the index pulse and the delivery of bits from the disk. This is caused by the time it takes to get DMA started. The delay ranges from 135–200 microseconds; at 4 microseconds to a bit, there are from 4 to 7 bytes of delay. 55 microseconds are used for software interrupt overhead, which is the time from the occurrence of the interrupt to the writing of the disk system DSKLEN hardware register. 66 microseconds are used for a horizontal scan-line delay, which synchronizes the disk I/O with the Agnus chip-display scan-line fetches. An additional scan-line time of 0–65 microseconds can be required; the DSKLEN register can be initialized anywhere along the scanning distance of one horizontal line.

## TD_RAWWRITE

### Purpose of Command

This command writes raw data bits to a unit of the TrackDisk device. TD_RAWWRITE transfers data bits from a task-defined buffer to the disk with no Blitter preprocessing. It is used to write raw data to a disk where there is no need to work with disk-change or sector-label information.

### Preparation of the IOExtTD 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 Unit structures that manage the addressed unit of the TrackDisk device; these parameters can be copied from the IOExtTD structure initialized by OpenDevice. Set io_Command to TD_RAWWRITE. Also set the following command-specific parameters:

- io_Flags. Set this to 0, or set it to IOTDF_INDEXSYNC if you want the Track-Disk device to try to write the track data from the index mark on the track. It may or may not be successful; however, keep in mind that there will always be a delay—perhaps a very long delay if, for example, interrupts have been disabled.
- io_Length. Set this to the length of the task-defined buffer. The maximum length is 32K.
- io_Data. Set this to point to the task-defined buffer from which the raw data bits will be copied. The buffer must be in chip memory (MEMF_CHIP).
- io_Offset. Set this to the number of the track to which the data bits should be written.

### Discussion

TD_RAWWRITE is used for write operations that do not use the last two parameters in the IOExtTD structure. No preprocessing of the data will occur during data transfer; the data bits written on disk will be exactly as they were in the task-defined buffer. Because the data will be arranged in MFM format, you should use this command only if you know how MFM data is defined and used. In addition, if you use TD_RAWWRITE, your program may not be compatible with future releases of Amiga software.

## *TD_REMCHANGEINT*

### Purpose of Command

This command removes a disk-change software interrupt from the TrackDisk device software system. It unlinks the Interrupt structure originally added by an TD_ADDCHANGEINT command and sends a reply to the task reply-port queue.

### Preparation of the IOExtTD 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 Unit structures that manage the addressed unit of the TrackDisk device; these parameters can be copied from the IOExtTD structure initialized by OpenDevice. Also set io_Command to TD_REMCHANGEINT, and set io_Flags to 0. Initialize io_Data to point to the Interrupt structure for the disk-change interrupt you want removed.

### Discussion

The TD_REMCHANGEINT and TD_ADDCHANGEINT commands allow the TrackDisk device software system to support an extendable list of disk-change software interrupts. These two functions allow you to customize disk-change operations to suit your own needs.

## *TD_REMOVE*

### Purpose of Command

This command allows a task to execute a disk-change software interrupt. It allows a task to take specific actions when a user inserts or removes a disk from a disk drive.

### Preparation of the IOExtTD 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 Unit structures that manage the addressed unit of the TrackDisk device; these parameters can be copied from the IOExtTD structure initialized by OpenDevice. Set io_Command to TD_REMOVE, and set io_Flags to 0. Initialize io_Data to point to the Interrupt structure that will manage the interrupt; this structure can be added with the TD_ADDCHANGEINT command.

### Discussion

TD_REMOVE is responsible for calling a specific disk-change interrupt routine whenever a disk is inserted or removed from a disk drive. By designing your own interrupt routines, you can arrange for various actions when a disk is changed. Each Interrupt structure contains a pointer to the data and code that will be used to respond to the disk interrupt, and the Track-Disk device internal routines will call the interrupt routine whenever a disk is inserted or removed. If io_Data is specified as null, disk-change interrupts will be suspended.

## *TD_SEEK*

### Purpose of Command

This command allows a task to move the disk-drive heads, thereby controlling their next read or write position. The IOExtTD structure io_Offset parameter specifies the desired seek location on the disk. TD_SEEK does not read any data, and it does not determine if the disk has been changed.

### Preparation of the IOExtTD 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 Unit structures that manage the addressed unit of the TrackDisk device; these parameters can be copied from the IOExtTD structure initialized by OpenDevice. Set io_Command to TD_SEEK, and set io_Flags to 0. Set io_Offset to the the desired byte-offset seek position on the disk.

### Discussion

Disk-drive heads automatically move to appropriate locations during the execution of TrackDisk device commands; no explicit TD_SEEK command is necessary. However, with TD_SEEK, a task can cause the disk-drive heads to move to a track and sector location for the next read or write operation. This "preseek" capability provides additional speed in disk access for both read and write operations. TD_SEEK will not verify its position until the next read operation. Your tasks can also use the ETD_SEEK command if they need to prevent a seek operation when a disk is changed.

# Appendix

## C Language Definitions of the Exec-Support Library Functions

This appendix presents the C language definitions of the Exec-support library functions, which were discussed in Chapter 2. Because the NewList function calls an assembly language macro, it is not included here.

Most often, the Exec-support library functions are used for task–device management. However, they also have general uses in any program that deals with tasks, message or reply ports, standard or extended I/O request structures, and lists. If you study the C language listings presented here, you can use them as templates for your own C language programs.

The Exec-support library functions must be located in the LIB: directory amiga.lib file on the C language programming disk in the external disk drive, as discussed in the introduction to this book. In contrast to other library files, these files are not called by placing their arguments into 68000 CPU registers. Instead, you place their arguments onto the task's stack and then call the function in the usual C language manner.

Structures associated with these functions (MsgPort, IOStdReq, and IOExtReq) are allocated with the Exec library AllocMem function using the MEMF_PUBLIC and MEMF_CLEAR memory attributes. MEMF_PUBLIC ensures that the structure is allocated in public RAM, and MEMF_CLEAR ensures that all structure parameters are initialized to 0 when the memory is allocated.

## CreateExtIO

The listing that follows shows the program statements that are necessary to define the CreateExtIO function. It shows you how to define the parameters in the IORequest structure and how to allocate memory for it using the AllocMem function. It also illustrates how you can error-check for available memory, typecast C language variables, and specify parameters in a structure that has nested substructures (the IORequest structure Message substructure has a Node substructure).

```
struct IORequest *CreateExtIO (taskReplyPort, size_extreq)
   struct MsgPort *taskReplyPort;
   LONG size_extreq;
{
   struct IORequest *myExtReq;
   if (taskReplyPort == 0)
      return ((struct IORequest *) 0);
   myExtReq = (struct IORequest *) AllocMem (size_extreq,
         MEMF_CLEAR | MEMF_PUBLIC);
   if (myExtReq == 0)
      return ((struct IORequest *) 0);
   myExtReq->io_Message.mn_Node.ln_Type = NT_MESSAGE;
   myExtReq->io_Message.mn_Node.ln_Pri = 0;
   myExtReq->io_Message.mn_ReplyPort = taskReplyPort;
   return (myExtReq);
}
```

## CreatePort

The following listing shows the program statements that are necessary to define the CreatePort function. It shows you how to define the parameters in the MsgPort structure and how to allocate memory for it using the AllocMem function. It also shows you how to error-check for available memory, typecast C language variables, allocate and free a signal bit number for a message port, and find the task that owns that port. It also illustrates how parameters are specified in a structure that has nested substructures (the MsgPort structure has a Node substructure), how message ports are added to the system message-port list, and how to create a new list.

```
struct MsgPort *CreatePort (msgPortName, msgport_priority)
  char *msgPortName;
  BYTE msgport_priority;
{
  UBYTE msgport_signalbitnumber;
  struct MsgPort *myMsgPort;
  if ((msgport_signalbitnumber = AllocSignal (-1)) == -1)
    return ((struct MsgPort *) 0);
  myMsgPort = AllocMem ((ULONG) sizeof(*myMsgPort),
      MEMF_CLEAR | MEMF_PUBLIC);
  if (myMsgPort == 0) {
    FreeSignal (msgport_signalbitnumber);
    return ((struct MsgPort *) (0)); }
  myMsgPort->mp_Node.ln_Name = msgPortName;
  myMsgPort->mp_Node.ln_Pri = msgport_priority;
  myMsgPort->mp_Node.ln_Type = NT_MSGPORT;
  myMsgPort->mp_Flags = PA_SIGNAL;
  myMsgPort->mp_SigBit = msgport_signalbitnumber;
  myMsgPort->mp_SigTask = FindTask (0);
  if (msgPortName != 0)
    AddPort (myMsgPort);
  else
    NewList (&(myMsgPort->mp_MsgList));
    return (myMsgPort);
}
```

## CreateStdIO

The following listing shows the program statements that are necessary to define the CreateStdIO function. It shows you how to define the parameters in the IOStdReq structure and allocate memory for it using the AllocMem function. It illustrates how you can error-check for available memory, typecast C language variables, and specify parameters in a structure that has nested substructures (the IOStdReq structure has a Message substructure that has a Node substructure).

```
struct IOStdReq *CreateStdIO (taskReplyPort)
  struct MsgPort *taskReplyPort;
{
  struct IOStdReq *myStdReq;
  if (taskReplyPort == 0)
    return ((struct IOStdReq *) 0);
  myStdReq = AllocMem (sizeof(*myStdReq), MEMF_CLEAR
      | MEMF_PUBLIC);
  if (myStdReq == 0)
    return ((struct IOStdReq *) 0);
  myStdReq->io_Message.mn_Node.ln_Type = NT_MESSAGE;
  myStdReq->io_Message.mn_Node.ln_Pri = 0;
  myStdReq->io_Message.mn_ReplyPort = taskReplyPort;
  return (myStdReq);
}
```

## CreateTask

The following listing shows the program statements that are necessary to define the CreateTask function. It shows you how to define some of the Task structure parameters and allocate memory using the AllocMem function. It also shows you how to error-check for available memory, set a task's stack, typecast C language variables, and specify parameters in a structure that has substructures (the Task structure has a Node substructure). Finally, it illustrates how you can add a task to the system task list with AddTask.

```
struct Task *CreateTask (myTaskName, myTaskPriority,
      task_EntryPoint, task_stacksize)
  char *myTaskName;
  UBYTE myTaskPriority;
  APTR task_EntryPoint;
  ULONG task_stacksize;
{
  struct Task *myTask;
  ULONG dataSize = (task_stacksize & 0xFFFFFC) +1;
  myTask = AllocMem ((ULONG) sizeof (*myTask) + dataSize,
      MEMF_CLEAR | MEMF_PUBLIC);
  if (( !(ULONG) myTask) {
    return ((struct Task *) (0)); }
    myTask—>tc_SPLower = (APTR) ((LONG)myTask + (LONG)
        sizeof(*myTask));
    myTask—>tc_SPUpper = (APTR) ((ULONG) (myTask—>
        tc_SPLower + dataSize) & 0xFFFFFE);
    myTask—>tc_SPReg = (APTR) ((LONG) (myTask—>tc_SPUpper));
    myTask—>tc_Node.ln_Type = NT_TASK;
    myTask—>tc_Node.ln_Pri = myTaskPriority;
    myTask—>tc_Node.ln_Name = myTaskName;
    AddTask (myTask, task_EntryPoint, 0);
    return (myTask);
}
```

## *DeleteExtIO*

The following listing shows the program statements that are necessary to define the DeleteExtIO function. It shows you how to change parameters in a structure and deallocate the memory assigned to the IORequest structure.

```
DeleteExtIO (myExtReq, size_extreq)
    struct IORequest *myExtReq;
    LONG size_extreq;
{
    myExtReq—>io_Message.mn_Node.ln_Type = 0xFF;
    myExtReq—>io_Device = (struct Device *) -1;
    myExtReq—>io_Unit = (struct Unit *) -1;
    FreeMem (myExtReq, size_extreq);
}
```

## *DeletePort*

The following listing shows the program statements that are necessary to define the DeletePort function. It shows you how to remove a message port from the system message port list, free its signal bit number, and deallocate the memory it had been assigned.

```
DeletePort (myMsgPort)
    struct MsgPort *myMsgPort;
{
    if ((myMsgPort—>mp_Node.ln_Name) != 0)
       RemPort (myMsgPort);
    myMsgPort—>mp_Node.ln_Type = 0xFF;
    myMsgPort—>mp_MsgList.lh_Head = (struct Node *) -1;
    FreeSignal (myMsgPort—>mp_SigBit);
    FreeMem (myMsgPort, (ULONG) sizeof(*myMsgPort));
}
```

## DeleteStdIO

The following listing shows the program statements that are necessary to define the DeleteStdIO function. It shows you how to change parameters in a structure and deallocate the memory assigned to that structure.

```
DeleteStdIO (myStdReq)
   struct IOStdReq *myStdReq;
{
   myStdReq—>io_Message.mn_Node.ln_Type = 0xFF;
   myStdReq—>io_Device = (struct Device *) -1;
   myStdReq—>io_Unit = (struct Unit *) -1;
   FreeMem (myStdReq, sizeof (myStdReq));
}
```

## DeleteTask

The following listing shows the program statements that are necessary to define the DeleteTask function. It shows you how to remove a task from the system task list and deallocate the memory assigned to the Task structure.

```
DeleteTask (myTask)
   struct Task *myTask;
{
   RemTask (myTask);
   FreeMem (myTask, 1 + (ULONG) (myTask—>tc_SPUpper)
   - (ULONG)myTask);
}
```

# Index

A
AbortIO function, 40–41
Access mode, *xxv*
  programming procedures and, 29
  shared versus exclusive, 8–10
ADCMD_ALLOCATE command, 57, 60, 61, 64, 69, 79–81
ADCMD_FINISH command, 62, 81–82
ADCMD_FREE command, 60, 68, 82–83
ADCMD_LOCK command, 83–84
ADCMD_PERVOL command, 62, 65, 85–86
ADCMD_SETPREC command, 60, 86–87
ADCMD_WAITCYCLE command, 62, 87–88
AddDevice function, 42–44
AddTime function, 311–312
Alink program, *xxx*
Allocation key, 60–61
Amiga computers, *xxi*
  capabilities of, *xxiv*
  programming environment for, *xxvii–xxxi*
  releases for, 204, 335, 343, 344
AmigaDOS, 246
  Keyboard device processing and, 222, 223, 226
Amplitude modulation, 56, 57, 58
ANSI character conversion, 208–210
Anti-aliasing filter, 65
Arithmetic, time, 305
ASCII characters, 119, 193, 213
  Intuition windows and, 194
Assembly language. See C language
Asynchronous I/O requests
  BeginIO and, 4, 41–42
  processing of, 10–12, 30–32
  SendIO and, 4
AudChannel structure, 63, 66
Audio device, 13, 55–88
  access modes, 10
  allocation key, 60–61
  BeginIO and, 42
  channels allocation, 56–60
  error codes, 66–67
  hardware configuration, 55–56
  I/O request structure, 23, 24

phoneme strings, 91
speech narration requests to, 109–111
Audio device commands, 15, 61–63, 70–79
aborting write command, 81–82
allocating channel groups, 79–81
changing channel precedence, 86–87
changing period and volume, 85–86
clearing internal buffers, 70–71
clearing/restarting channels, 73–74
completing waveform cycle, 87–88
device-specific, 79–88
flushing I/O requests, 71–72
freeing/reallocating channels, 82–83
locking channels, 83–84
playing information to hardware, 77–79
programming process with, 63
reading a channel pointer, 72–73
standard, 70–79
starting channels, 74–75
stopping channels,75–76
writing buffers to hardware, 76–77
Audio device functions, 67–70
Audio device structures, 63–66

**B**

BeginIO function, *xxv,* 3, 40, 41–42, 210. *See also* Internal processing
Bits
Audio device, 55
keyboard keys and, 234
parallel-port, 116, 118
serial-port status register, 138, 139
*see also* Flag parameter bits
Blitter coprocessor, 324, 335, 336
Blocks of data. *See* Data blocks
BootBlock structure, 330
Break signals, 158–159
Buffers, *xxv–xxvi*
commands for internal device, 14–16
double, 55
internal device, *xxv,* 3
Parallel device, 115, 116, 127
programmer-defined, 13
Serial device, 137
Buffers, task-defined, *xxv,* 2, 127, 131
with multiple ports, 6
read–write operations for, 115
Bytes, 234

**C**

C language,
Exec library function definitions in, 351–355
Lattice compiler for, *xxvii*
programming procedures for, 27–30
C: directory, *xxx*
Caches, 16
CBD_CLIPREADID command, 298–299
CBD_CLIPWRITEID command, 299
CBD_CURRENTREADID command, 284, 301
CBD_CURRENTWRITEID command, 284, 301
CBD_POST command, 284, 296, 297, 298–299, 300–301
CDInputHandler function, 197
Console device, 204–205, 207, 209
CD_ASKDEFAULTKEYMAP command, 214–215, 216
CD_ASKKEYMAP command, 215–216
CD_SETDEFAULTKEYMAP command, 216–217
CD_SETKEYMAP command, 217–218
Channel allocation, 56–60
Channel allocation precedence, 59–60
Channel combination array, 57–59
number of entries in, 100
Character termination array, 115, 119, 127, 131
CMD_START versus, 155
CMD_STOP versus, 156
definition of, 134
parameters for, 162–163
Serial device and, 137,
Serial device structure and, 141–142
Characters
ANSI actions and, 213
control-code, 259, 261–262
screen control, 213, 214
Chip, complex interface adapter (CIA), 306
Cleanup routines, 221
Clip identifiers, 384
Clipboard device, 13, 283–301
I/O request structure and, 23, 24
operation of, 283–286
sequential read–write operations, 284–286
Clipboard device commands, 15
determining read clip identifier, 298–299
determining write clip identifier, 299
device-specific, 298–301
posting clip, 300–301
programming process for, 286–287
reading characters into buffer, 293–294
resetting unit to boot-up, 295
standard, 293–298
updating internal routines, 295–296
writing characters from buffer, 296–298
Clipboard device functions, 290–293
Clipboard device structures, 287–290
preparing, 286
ClipboardUnitPartial structure, 287, 288
CloseDevice function, *xxv,* 16
Audio device, 67–68
Clipboard device, 290–291
Console device, 205–206
Gameport device, 245
Input device, 175–176
Keyboard device, 225–226

Narrator device, 98
Parallel device, 121–122
Printer device, 268–269
RemDevice, 43–44
Serial device, 146–147
Timer device, 312–313
TrackDisk device, 332
CloseLibrary function, 99
CMD_CLEAR command, 14, 15
Audio device, 70–71
Console device, 210–211
Gameport device, 248
Keyboard device, 228–229
Parallel device, 115
Serial device, 149
CMD_FLUSH command, 5, 14, 15
Audio device, 71–72
Input device, 178
Narrator device, 92, 104–105
Parallel device, 124–125
Printer device, 263, 271–272
Serial device, 149–150
CMD_INVALID, 14, 15
CMD_POST command. See CBD_POST command
CMD_READ command, 14, 15
Audio device, 72–73
Clipboard device, 284–286, 293–294
Console device, 193–197, 210,
Console device, 211–212
Narrator device, 92, 105–106
Parallel device, 115, 116, 125–127
Serial device, 150–152
CMD_RESET command, 5, 14, 15
Audio device, 73–74
Clipboard device, 295
Narrator device, 92, 107
Parallel device, 127–128
Printer device, 272
Serial device, 153
CMD_START command, 14, 15
Audio device, 74–75
Input device, 179–180
Narrator device, 92, 107–108
Parallel device, 128–129
Printer device, 263, 272–273
Serial device, 154–155
CMD_STOP command, 14, 15
Audio device, 75–76
Input device, 180
Narrator device, 92, 108–109
Parallel device, 115, 129
Printer device, 263, 273–274
Serial device, 155–156
CMD_UPDATE command, 15, 16
Audio device, 76–77
Clipboard device, 295–296
CMD_WRITE command, 15, 16
Audio device, 77–79
Clipboard device, 284, 286, 296–298
Console device, 193–197, 212–214
Narrator device, 92, 109–110
Parallel device, 115, 116
Printer device, 263, 271, 274
Serial device, 156–158
CmpTime function, 313–314
Command-line interface (CLI), 259
Compile-and-link script file, *xxx, xxxi*
Complex interface adapter (CIA) chip, 306
Connectors. *See* Pin connections
Console device, 13, 167, 193–218
default key map and, 214–215
I/O request structure for, 23, 24
mapped data and, 216–218
opening Gameport device units and, 246
raw key conversion for unit of, 208–210
read-write operations and, 193–197
Timer device and, 305
Console device commands, 15, 197–198
clearing read buffer, 210–211
copying current key map parameters,215–216
copying data into current key map, 217–218
copying data into default key map, 216–217
copying default key map parameters, 214–215
device-specific, 214–218
programming process for, 197–198
reading characters into buffer, 211–212
standard, 210–214
writing characters from buffer, 212–214
Console device functions, 204–210
closing unit access, 205–206
converting raw key codes, 208–210
handling input events, 204–205
opening routine access, 206–208
Console device structures, 198–204
preparing, 197
Control sequence introducer (CSI), 214
Control-code characters, 259, 261–262
Control-code files, 262
Control-Q command, 128, 129
Control-S command, 129
Controllers, 239
Gameport, 239, 241, 249–250, 253–254
proportional, 241
ConUnit structure, 193, 194
Console device and, 198–202
Console OpenDevice function and, 207
pointer parameters, 199
Copy operations, 38
CreateExtIO function, 44–45
Printer device and, 265, 270
programming statement for, 351
CreatePort function, 45–46

Console device and, 193
programming statement for, 352
CreateStdIO function, 46–47
programming statement for, 352–353
CreateTask function, 47–48
programming statement for, 353–354
Custom structure, 63

D

Data blocks, 131, 134, 152
Serial device and, 137, 158
Data register, parallel-port, 115–116, 127
Data transfer
device-to-device, 13
parameters for, 143–144
from Serial device, 158
Debugging file, *xxxi*
Decibels, 55, 65
Default values
device access to, 9
Gameport device unit access mode, 247
key map, 214–215
Keyboard device and, 228
map, 196, 197
Narrator device and, 107
DeleteExtIO function, 48–49
programming statement for, 354
DeletePort function, 49
programming statement for, 354–355
DeleteStdIO function, 50
programming statement for, 355
DeleteTask function, 50–51
programming statement for, 355
Device(s)
Exec functions for, 16
internal buffer, 3
internal routines, 1, 3
multiple, 6
programming concepts for, *xxi*
request and reply-port queues, 4, 5
ROM/disk resident, *xxiv*
shared versus exclusive access, 8–10
task interactions with, *xxi*, *xxv*, 1–3, 12–13
unit I/O request port and, 3
Device functions, 16
Device I/O requests, 1–24
MsgPort structure for, 18
*see also* Immediate-mode request processing; Queued I/O requests; Quick I/O requests
Device internal routines, *xxi*
Console versus Intuition, 194–195
data transfer and, 13
immediate-mode commands and, 35
Parallel device, 116
programming procedures for, *xxiv–xxvii*
queued I/O and, 3
quick I/O and, 5
Device library, 1
Device library functions, 40–44
Device management, 27–52
Device message-port queues, 8
Device structure, *xxv*, 2, 3
Device unit(s)
allowed, 13
opening and closing of, 29
request queue processing with, 31
DEVS: directory, *xxix*
Direct memory access (DMA) channel, 55, 56
Directories, *xxix–xxxi*
DoIO function, *xxv*, 3, 40–41
BeginIO versus, 41–42
*see also* Internal processing
Dvorak keyboard, 197

E

End-of-file (EOF) characters, 125, 134
English strings, 103–104
processing of, 91, 94
Error codes, *xxvi–xxvii*
access operations and, 9
Audio device, 66–67
common device, 67
Narrator device, 97–98
Parallel device, 121
Serial device, 140, 145
TrackDisk device, 330–331
ETD_CLEAR command, 326, 333–334
ETD_FORMAT command, 326, 340
ETD_RAWREAD command, 324, 334–335
ETD_RAWWRITE command, 335–336
ETD_READ command, 324
ETD_UPDATE command, 326, 336–337
Exclusive mode access, 8–10
Exec library functions, 16, 42–52
AddTask/FindTask, 27
file containing, *xxxi*
programming statements for, 351–355
ExecWait function, 4
Expunge routines, 42–43, 122
Narrator device and, 100

F

FD.FILES: directory, *xxxi*
First-in, first-out (FIFO), 31
Flag parameter bits, *xxvii*, 8, 9
bit values in the I/O structure, 65
file containing, *xxxi*
general programming and, 29
I/O request structures and, 19, 37
MsgPort structure and, 22
parallel-port, 120–121
Serial device, 8–9, 143–145

Unit structure, 21
Floppy disks
supported by Amiga, 323
TrackDisk device interactions with, 325-326
Floppy-disk drives, 16
FONTS: directory, *xxx*
Frequency modulation, 56, 57, 58
Functions. *See* Exec library functions; specific device functions

G
Gameport device, 13, 193, 239-255
buffers for, 240, 248, 253
hardware connected to, 240
as input event source, 168
I/O request structure of, 23, 24
Input device and, 167
input event processing and, 240, 242
Keyboard device and, 221
operation of, 240-242
units of, 240-242
*see also* Console device
Gameport device commands, 15
changing controller type, 253-254
changing current trigger conditions,255
clearing internal read buffer, 248
determining current trigger conditions, 250-251
determining controller type, 249-250
device-specific, 249-255
programming process, 243-244
reading input events into buffer,251-253
standard device, 248
Gameport device functions, 245-248
Gameport device structures, 244
preparing, 243
GamePortTrigger structure, 240, 244
GetMsg function, 3, 4
GPD_ASKCTYPE command, 240, 249-250
GPD_ASKTRIGGER command, 240, 250-251
GPD_READEVENT command, 240, 247, 251-253
GPD_SETCTYPE command, 240, 249, 253-254
GPD_SETTRIGGER command, 240, 255
Graphics library
parameters in, 201-202
phoneme strings and, 91, 94

H
Handshaking protocol, 128, 129, 154-155, 156, 163
Hard-disk drives, 16

I
I/O requests, *xxv*, 1
access modes and, 8-10
classes of, 37
device classes, 3-5
initializing and processing, 35-37
multiple ports and units and, 5-6
multiple, 37-40
multitasking and, 10-12
ports, 2
processing of, 10-12
queue behavior and, 6-8
structure linkages and, 16-18
task-device interactions in, 1-3
Immediate-mode operations, 34-35
Clipboard device, 286
Console device, 197
Gameport device, 243
Input device, 171
Keyboard device, 224
Parallel device, 116
Timer device, 309
INCLUDE files, *xxi, xxxi,* 23, 24, 37, 242
IND_ADDHANDLER command, 181-182
IND_REMHANDLER command, 182
IND_SETMPORT command, 183
IND_SETMTRIG command, 184
IND_SETMTYPE command, 184-185
IND_SETPERIOD command, 186-187
IND_SETTHRESH command, 187-188
IND_WRITEEVENT command, 188-189
Initialization of device-specific I/O request structures, 29
Input device, *xxiv,* 13, 167-189
access mode of, 176
Console device and, 193
event handling by, 167
Gameport device dedications and, 240
Gameport input events and, 253
I/O request structure and, 23, 24
internal routines of, 168, 170
Keyboard device and, 221
opening Gameport device units and, 246
operation of, 168
software system and, 168
Timer device and, 305
TrackDisk device and, 325
Input device commands, 15
aborting queued requests, 178
adding input-handler functions, 181-182
defining input events by task, 188-189
device-specific, 181-189
establishing device type at mouse connector, 184-185
establishing threshhold time period, 187-188
establishing time interval for repeating key, 186-187
establishing mouse input trigger conditions, 184
programming process and, 171-172
removing input-handler functions, 182
resetting unit 0, 178-179
restarting unit 0, 179-180
specifying gameport by task, 183
standard, 178-180

stopping command execution, 180
Input device functions, 175-177
Input device structures, 172-175
preparing, 171
Input event stream, 167, 168
Console device and, 193, 196
InputEvent structure, 167, 170-171, 172, 173-175
Keyboard device and, 221
Input-handler functions, 167, 168-171
design of, 170-171
priority parameters, 170
Input task, 167
INQ-ACQ handshaking protocol, 155
Internal routines. *See* Device internal routines
Interrupt structure, 172-173
Intuition software system, 170
Intuition windows, 13, 16
Console device and, 193-197, 207
Input device and, 167
parameters for, 199
*see also* Console device
IO requests. *See* I/O requests
IOAudio structure, 63-65
IOClipReq structure, 284, 288-290
IODRPReq structure, 265, 267-268
IOExtPar structure, 2
Parallel device, 119, 120-121
IOExtSer structure, 2
Serial device, 141, 142-145
IOExtTD structure, 328-329
IOF_QUICK, 3, 4
IOPar structure, 119
IOPArray structure, 119
IOPrtCmdReq structure, 265, 266-267
IORequest structure, 2
definition of, 18
parameters in, 19
IOSer structure, 141, 142-143
IOStdReq structure, 2, 17-18, 37
Console device and, 197
definition of, 20
parameters in, 19-20
IOTArray structure, Serial device, 141-142
io_Data parameter, 2

J

Joysticks
absolute, 241
relative, 241

K

KBD_ADDRESETHANDLER command, 221, 224, 230-231, 235
KBD_READEVENT command, 222, 228, 229, 230, 231-233
KBD_READMATRIX command, 221, 223, 233-234
KBD_REMRESETHANDLER command, 221, 235
KBD_RESETHANDLERDONE command, 221, 236
Key map, 196, 197
Keyboard device, 13, 193, 221-236
as input event source, 168
buffers of, 221
event stream to Console device and, 196
I/O request structure for, 23, 24
Input device and, 167
input event processing, 222-224
key definition and, 234
operation of, 221-222
preprocessing of events, 168
units of, 221
Keyboard device commands, 15
adding reset-handler function, 230-231
clearing internal read buffer, 228-229
device-specific, 230-236
programming process with, 224
reading current key matrix into buffer, 233-234
reading keystrokes into buffer, 231-233
removing reset-handler function, 235
resetting device to boot-up, 229-230
standard, 228-230
terminating execution of a reset-handler function, 236
Keyboard device functions, 225-228
Keyboard device structures, 225
for Console device, 198
Keyboard device structures, 224
Keyboards, Qwerty/Dvorak, 197
KeyMap structure, 198, 199, 201, 202-203, 214-215
KeyMapNode structure, 199, 203
KeyMapResource structure, 199, 204
Kickstart disk, *xxviii-xxix*

L

L: directory, *xxix*
Lattice C language compiler, *xxvii*
LIB: directory, *xxx-xxxi*
Library function. *See* Device library functions; Exec library functions
Library structure, 2, 3
LIBS: directory, *xxix*
lib_OpenCnt parameter, *xxv,* 10
List management, 3
List substructure, 18
Locking channels, 59-60

M

Maps, 196, 197
Memory, *xxiv*
automatic closing and, 16
Input events and, 167
RemDevice function and, 43
task-device relationships and, 12-13
*see also* RAM

Message ports, 3, 28
  Console device and, 193, 195
  *see also* Task reply ports
Message structures, 1, 17, 18
  definition of, 22
  parameters, 22
Modem, 137, 160
Modified frequency modulation (MFM) format, 335, 343, 344
Mouse, 241
  Console device and, 193, 196
  gameport controller and, 239
  Gameport input events, 197
  as input event source, 167, 168
Mouth shape, 91, 105–106
  parameters for, 96
  Voice substructure and, 96–97
Mouth_rb structure, 96–97
MsgPort structure, 1, 17, 18
  definition of, 21
  parameters in, 22
  signals and, 4
Multitasking, 10–12
  Exec library functions and, 33–34
Music programs, input events and, 196
Musical instrument digital interface (MIDI), 163

N

N-key typeahead, 222
Narrator device, 13, 91–111
  error codes, 97–98
  I/O request structure of, 23, 24
Narrator device commands, 15, 104–111
  aborting I/O requests, 104–105
  displaying mouth shapes, 105–106
  narrating phoneme strings, 109–111
  programming process, 92–93
  resetting unit to default, 107
  restarting unit, 107–108
  standard, 104–111
  stopping write commands, 108–109
Narrator device functions, 98–104
  closing Translator library access, 99
  closing unit access, 98
  opening Translator library access, 102–103
  opening unit access, 99–102
  translating English strings, 103–104
Narrator device structures, 94–97
Narrator_rb structure, 94–96
NewList, 51–52
Node substructure, 16

O

OpenDevice function, *xxiv–xxv,* 3, 16, 246–248
  Audio device, 68–70
  Clipboard device, 291–293
  Console device, 193, 206–208, 209, 213
  Input device, 176–177
  Keyboard device, 226–228
  Narrator device, 99–102
  Parallel device, 122–124
  Printer device, 269–271
  RemDevice, 43–44
  Serial device, 147–149
  Timer device, 314–316
  TrackDisk device, 332–333
OpenLibrary function, 102–103
Ordinal clip identifiers, 284

P

Parallel device, 13, 115–134
  access modes in, 122, 124
  data register, 130
  error codes for, 121
  I/O request structures and, 2, 23, 24
  Printer device and, 269, 270
  read–write operations, 115–116
  Timer device and, 305
Parallel device commands, 15, 116
  aborting I/O requests, 124–125
  device-specific, 131–134
  programming process, 117–118
  querying routine status, 131–133
  reading data into buffer, 125–127
  resetting/restarting unit, 127–128
  restarting reads and writes, 128–129
  setting parameters by task, 133–134
  standard, 124–131
  stopping read/write execution, 129
  writing from buffer, 130–131
Parallel device functions, 121–124
Parallel device structures, 117–118, 119–121
Parallel-port connector, 116
Parallel-port data register, 115–116, 127
Parallel-port status register, 116, 118
Parameter processing
  Console device commands, 198
  Gameport device commands, 243–244
  Input device commands, 171–172
  Keyboard device commands, 224–225
  Narrator device commands, 93
  Parallel device commands, 118
  Printer device commands, 264–265
  Serial device commands, 139–141
  Timer device commands, 309
  TrackDisk device commands, 327
Parameters
  allocation key and, 60–61
  ConUnit, 194, 195
  message structure, 22
  port determination and, 4
  RastPort structure, 195

reply port specification by, 6
SRE and RRE, 196
structure, 10
Unit structure, 21
*see also* discussions of commands
PDCMD_QUERY command, 116, 131–133
PDCMD_SETPARAMS command, 133–134
Period register, 56
Phoneme strings, 91, 103–104
Pin connections
external disk drives, 324
Gameport, 239, 240–241
parallel-port, 116, 117
serial-port, 138
Pitch of sound, 55
Pointer parameter, 6
Ports. *See* Reply ports; Task reply ports
Power failures, 77, 337
update operations and, 326
PRD_COMMAND command, 271, 274
PRD_DUMPRPORT command, 267, 271, 274, 275–
PRD_PRTCOMMAND command, 262, 263, 266
PRD_RAWWRITE command, 262, 271, 274, 279–28
Printer command files, 262
Printer device, 13, 259–280
control codes and, 261–262
I/O request structure for, 23, 24
operation of, 259–261
PrinterIO union and, 265, 270–271
programming process, 264–265
Printer device commands, 15
aborting I/O requests, 271–272
device-specific, 275–280
printing raster bitmap, 275–277
resetting unit O, 272
restarting, 272–273
standard, 271–275
stopping execution, 273–274
transmitting series of control codes, 279–280
transmitting single control-code commands, 278–279
transmitting control-code characters, 274–275
Printer device functions, 268–271
Printer device structures, 264, 265–268
PrinterIO union, 265, 270–271
Printers, supported by Amiga, 259
Priorities, in channel allocation, 56–61
Programming procedures, *xxi, xxiv–xxvii*
Programming. *See* C language

Q
Quad file, *xxx*
Queued I/O requests, 3–4
Clipboard commands, 286
Console device commands, 197, 198
Gameport device commands, 243
Input device commands, 171
Keyboard device commands, 224
Narrator device commands, 92
Parallel device commands, 116
restarting, 129
Timer device commands, 309
TrackDisk device commands, 324
*see also* Asynchronous I/O requests; Synchronous I/O requests
Queues
reply-port, 4
request, 4
*see also* Queued I/O requests
Quick I/O (QuickIO) requests, 3, 4–5
Audio device commands, 62
Clipboard device commands, 286
Console device commands, 197, 198, 210, 211, 214, 216
Gameport device commands, 243
Input device commands, 171
Keyboard device commands, 224
Narrator device commands, 92
Parallel device commands, 116
Printer device commands, 263
process decision, 1
Timer device commands, 309
TrackDisk device commands, 324
*see also* discussions of specific commands
Qwerty keyboard, 197

R
RAM, *xxiv, xxvii*
expansion kits for, *xxix*
InputEvent structures and, 167, 171, 172
internal/external, *xxviii*
raw key conversion and, 210
synchronous I/O request processing and, 32–33
unions and, 265
Raster bitmap, 275–277
RastPort structure, 214
parameters for, 201–202
Raw key codes, 221
conversion of, 208–210
RawKeyConvert function, 197, 207, 208–210, 214, 215
Read error, 151
Read operations, *xxv*
Read-write operations, *xxv,* 115–116
Console device and, 193–197
sequential, 284–286
Serial device and, 137–138
Register. *See* Data register; Period register; Status register
RemDevice function, 42–44
Reply ports, 4–5
multiple, 5–6
ReplyMsg function, 1, 3, 4
Request ports, 4
Reset raw events (RRE), 196, 211, 214
Reset-handler functions, 221–222, 230–231

completion of, 236
removal of, 235
ROM-based input task, 16
Root directory
external disk, *xxx*
internal disk, *xxix*

S

S: directory, *xxix*
Sampling period, 55
SatisfyMsg structure, 284, 287, 290
SDCMD_BREAK command, 158-159
SDCMD_QUERY command, 160
SDCMD_SETPARAMS command, 161-163
SendIO function, *xxv*, 3, 210
Serial device, 13, 137-163
access modes of, 137
error parameter values for, 145
flag parameters and, 8-9
I/O request structures and, 1-2, 23, 24
multiple ports and, 5
Parallel device versus, 137
parameter setting for, 146, 161-163
Printer device and, 269, 270
read-write operations, 137-138
Timer device and, 146, 147, 305
Serial device commands, 15
aborting I/O requests, 149-150
clearing internal buffers, 149
device-specific, 158-163
programming process with, 139-140
querying status by task, 160
reading characters into buffer, 150-152
resetting unit to initialization state, 153
restarting writes or reads, 154-155
sending break signal, 158-159
setting parameters by task, 161-163
standard, 149-158
stopping current reads or writes, 155-156
writing characters from buffer, 156-158
Serial device functions, 146-149
Serial device internal read buffer, 137
clearing, 149
default size, 153
reading characters into, 150-152
Serial device structures, 141-145
preparing, 139
Serial-port connector, 138
Serial-port status register, 138
Set raw events (SRE), 196, 211, 214
Shared mode access, *xxv*, 8-10
Signals, 4
break, 158-159
Quick I/O and, 4
Software libraries, *xxi*
Sound, 55
frequency of, 56
Speakers, channel allocation to, 57-61
Speech, 109-111
Narrator parameters for, 95-96
Voice substructure and mouth, 96-97
Startup-sequence script file, *xxvii, xxix*
Statements. *See* Tasks
Status queries
Parallel device, 131-133
Serial device, 139, 160
write protection and, 342
Status register
parallel-port, 116, 118, 130
serial-port, 138, 139
Stealing of channels, 59-60
Structure linkages, 16-18
Structures. *See* name of device
Substructures, 1, 4
SubTime function, 316
Synchronous I/O requests
BeginIO and, 41-42
device processing for, 10-12
DoIO and BeginIO functions, 4
processing of, 32-33
System input task, 176
System time, 306

T

T: directory, *xxx*
Task priority, 10
Task reply ports, 3
creation of, 28
multiple, 5-6
Task reply-port queues, 6-7, 18
queued I/O and, 4
quick I/O and, 4-5
requests and, 4-5
Task(s)
Console device and, 193
creation of, for device management, 27
device interactions with, *xxi, xxv, xxvii,* 1-3, 12-13, 27-35
loop sequence of, *xxv*
statements defining, 2
switching, 9, 10
TDU_PublicUnit structure, 329-330
TD_ADDCHANGEINT command, 337, 345
TD_CHANGENUM command, 338
TD_CHANGESTATE command, 338-339
TD_FORMAT command, 326, 339-340
TD_GETDRIVETYPE command, 327, 340
TD_GETNUMTRACKS command, 327, 340-341
TD_MOTOR command, 341-342
TD_PROTSTATUS command, 342
TD_RAWREAD command, 324, 342-343
TD_RAWWRITE command, 344

TD_REMCHANGEINT command, 345
TD_REMOVE command, 325, 345–346
TD_SEEK command, 326, 346–347
Termination characters. *See* Character termination array
Text editor, *xxx*
Text, input events and, 196
Time, system, 306
Timer device, 13, 122, 123, 193, 305–320
  I/O request structure, 23, 24
  Input device and, 167
  as input event source, 168
  intervals, 307–308
  operation of, 305–306
  resolution, 305, 307
  timing error adjustments, 308
  units of, 306–307
  *see also* Console device
Timer device commands, 15
  ascertaining current system time, 318–319
  device-specific, 317–320
  programming process for, 309–310
  setting current system time, 319–320
  timing operations, 317–318
Timer device functions, 305, 311–317
  adding time values, 311–312
  closing unit access, 312–313
  comparing times, 313–314
  opening a unit, 314–316
  subtracting time values, 316
Timer device structures, 310–311
  preparing, 309
TimeRequest structure, 173, 311
TimeVal structures, 168, 310–311
TrackDisk device, 10, 12, 13, 37, 323–347
  buffers, 324
  Console device and, 196
  error codes, 330–331
  floppy disk interactions with, 325–326
  I/O request structure, 23, 24
  Input device and, 167
  as input event source, 168
  input events to Console device and, 197
  Keyboard device and, 221
  multiple devices and, 6
  operation of, 323–325
  power failures and, 77
  units of, 323–324
TrackDisk device commands, 15
  ascertaining disk presence, 338-339
  determining write-protection status,342
  device-specific, 333–347
  executing disk-change interrupt, 345–346
  formatting a disk, 339–340
  handling disk-change interrupts, 337
  incrementing disk-change counter, 338
  marking specified track invalid, 333–334
  moving disk-drive heads, 346–347
  normal/extended, 326
  programming process, 326–327
  reading raw data bits from unit, 342–343
  reading raw data bits, 334–335
  removing disk-change interrupt, 345
  returning number of available tracks, 340–341
  returning drive type to calling task, 340
  turning motor on/off, 341–342
  writing current contents to disk, 336–337
  writing raw data bits, 335–336, 344
TrackDisk device functions, 327
Trackdisk device structures, 327–330
  preparing, 326
TrackDisk functions, 332–333
Translate function, 103–104
Translator library, 12, 13, 91, 94, 99, 102, 103
TranslatorBase structure, 102
Trigger conditions, 250–251, 255
TR_ADDREQUEST command, 317–318
TR_GETSYSTIME command, 318–319
TR_SETSYSTIME command, 319–320

**U**

Unit structure, *xxv*, 2, 17
  Console device and, 193
  definition of, 21
  parameters in, 21
Units, allowable, 13
unit_OpenCnt parameter, *xxv*, 10

**V**

Voice substructure, 96–97
Voice synthesis library, 91, 104
Volume values of sound, 55, 65

**W**

WaitIO function, 4
WaitPort function, 4
Waveform array, 55, 56, 64
Word boundary, 55
Word-exception table, 91
Word wrap, 213
Workbench disk, *xxvii*, 197
Write control store (WCS) memory, *xxiv*, *xxix*
Write error, 156
Write operation, *xxv*

# AMIGA PROGRAMMER'S HANDBOOK, VOLUME II

The **Amiga Programmer's Handbook, Volume II,** provides an in-depth reference to device I/O programming for the Amiga 500, 1000, and 2000 computers, including programming for sound and speech. Up-to-date, concise, and specially organized for the working programmer, this volume puts at your fingertips complete information on every command for the twelve Amiga devices, including commands that are new to the version 1.2 software release.

Two introductory chapters cover device I/O programming in general, with facts, functions, and techniques that apply to all the Amiga devices. Topics include device queueing, task queueing, and the Exec Support Library functions. Subsequent chapters are devoted to the individual devices:

- Audio
- Parallel
- Input
- Keyboard
- Printer
- Timer
- Translator/Narrator
- Serial
- Console
- Gameport
- Clipboard
- TrackDisk

Each chapter gives a detailed introduction to the device, followed by exhaustive reference entries describing its associated function calls and commands. The text is illustrated throughout with diagrams of device operations, command and function processing, and structure relationships in the software system—making this the most thorough and useful reference source you'll find anywhere.

This book is an absolute must for anyone seeking an understanding of Amiga devices, including professional programmers, programmer/owners who know C assembly language or any high-level language, and vertical-market applications developers who need in-depth information on Amiga device capabilities.

**Amiga Programmer's Handbook**
**Volume II**
**For Sound and Music**

**"Very highly recommended, no Amiga programmer should be without it."**
**—Commodore Magazine**

**About the Author**

Eugene P. Mortimore is president of Micro Systems Analysis, Inc., a microcomputer firm specializing in Amiga and IBM software and hardware support. He is the author of numerous books on microcomputer topics, and lives in Bethel Park, Pennsylvania.

SYBEX

ISBN 0-89588-384-8 **U.S. $24.95**