UNISYS

CTOS™

Programming Guide

Volume I
General Programming Topics

3.2 BTOS II
9.10 CTOS
2.4 CTOS/VM
3.0 CTOS/XE
12.0 Standard Software

Priced item

March 1990
Distribution code SA

Printed in USA
09-02392

UNISYS

CTOS®

Programming Guide

Volume I
General Programming Topics

Copyright © 1991 Unisys Corporation
All Rights Reserved
Unisys is a trademark of Unisys Corporation

CTOS I 3.3
CTOS II 3.3
CTOS/XE 3.0/3.1
Priced Item

June 1991
Printed in USA

43574490-110

The names, places, and/or events used in this publication are not intended to correspond to any individual, group, or association existing, living, or otherwise. Any similarity or likeness of the names, places, and/or events with the names of any individual, living or otherwise, or that of any group or association is purely coincidental and unintentional.

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THIS DOCUMENT. Any product and related material disclosed herein are only furnished pursuant and subject to the terms and conditions of a duly executed Program Product License or Agreement to purchase or lease equipment. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such License or Agreement. Unisys cannot accept any financial or other responsibility that may be the result of your use of the information or software material, including direct, indirect, special or consequential damages.

You should be careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Convergent, Convergent Technologies, CTOS, and NGEN are registered trademarks of Convergent Technologies, Inc.

Art Designer, AutoBoot, AWS, Chart Designer, ClusterCard, ClusterShare, Context Manager, Context Manager/VM, CTAM, CT-DBMS, CT-MAIL, CT-Net, CTOS/VM, CWS, Document Designer, Generic Print System, Image Designer, IWS, Network PC, PC Emulator, Phone Memo Manager, Print Manager, Series 186, Series 286, Series 386, Series 286i, Series 386i, Shared Resource Processor, Solution Designer, SRP, SuperGen, TeleCluster, The Operator, Voice/Data Services, Voice Processor, and X-Bus are trademarks of Convergent Technologies, Inc.

Intel is a registered trademark of Intel Corporation.

MS-DOS is a registered trademark of Microsoft Corporation.

UNISYS

Product Information Announcement

● New Release ○ Revision ○ Update ○ New Mail Code

CTOS™ Programming Guide: Volume I, General Programming Topics
Volume II, Extended System Services and Libraries

This Product Information Announcement announces the release and availability of the *CTOS Programming Guide* , Volume I: 09–02392, Volume II: 09–02393, dated April 1990. Information in this document is relative to BTOS II 3.2, CTOS 9.10, CTOS/VM 2.4, CTOS/XE 3.0, and Standard Software 12.0.

This Programming Guide describes techniques for effective programming in the CTOS environment. It concentrates on Unisys-specific hardware and system software programming. Both guides include specific programming examples. When ordered as a set, the Programming Guide includes a floppy disk containing the examples.

Volume I, General Programming Topics, provides an overview of programming in the Unisys CTOS/BTOS environment and includes specific sections on color programming, the Installation Manager, synchronous data communication, and writing distributed system services for the XE-530. Volume II, Extended System Services and Libraries, describes the use of system serivce procedures that are not part of the operating system but are distributed with it. Topics include Mouse Services, Voice/Data (Telephone) Services, Performance Statistics Services, Queue Manager, Spooler, and the writing of asynchronous system services using Async.lib.

You may order copies of this newly released Programming Guide as follows:

4120 5394–800	– Programming Guide Vols. I and II with binders and slipcases	
09–02392	– Programming Guide Volume I only	02392
09–02393	– Programming Guide Volume II only	02393

To order copies, contact your branch representative or:

Unisys Corporation, Corporate Software
and Publications Operations
13250 Haggerty Road
Plymouth, Michigan 48170

Please address all technical communication relative to this Programming Guide to:

Unisys Corporation
Multimedia Product Information
2700 N. First St.
P.O. Box 6685
San Jose, CA 95150–6685

CTOS is a trademark of Convergent, Inc

Announcement only:
SA

Announcement and attachments:

System: CTOS
Release: BTOS II 3.2, CTOS 9.10, CTOS/VM 2.4, CTOS/XE 3.0, Standard Software 12.0
April 1990
Part Number: 09–02392

Page Status

Page	Issue
Volume I	
i through xviii	6/91
1–1 through 1–17	6/91
1–18	Blank
2–1 through 2–13	Original
2–14	Blank
3–1 through 3–46	6/91
4–1 through 4–18	Original
5–1 through 5–79	6/91
5–80	Blank
6–1 through 6–9	Original
6–10	Blank
7–1 through 7–20	Original
8–1 through 8–13	Original
8–14	Blank
9–1 through 9–20	Original
10–1 through 10–10	Original
Index–1 through Index–22	6/91
Volume II	
i–xvi	6/91
1–1 through 1–30	Original
2–1 through 2–30	Original
3–1 through 3–15	Original
3–16	Blank
4–1 through 4–104	6/91
5–1 through 5–19	6/91
5–20	Blank
6–1 through 6–62	Original
7–1 through 7–58	6/91
8–1 through 8–28	6/91
Index–1 through Index–22	6/91

Contents

About This Manual

What Is the CTOS Programming Guide? xv
What's New in This Update xvi
Who Should Use This Guide xvi
Structure of This Guide xvi
Related Documentation xvii

1 The CTOS Programming Environment

Introduction 1-1
Distributed Client-Server Computing: The CTOS Cluster 1-1
Tools Available to the CTOS Programmer 1-2
An Overview of CTOS System Software 1-3
General Steps in Writing a CTOS Program 1-5
  Writing Source Code 1-5
  Compiling 1-6
  Linking 1-6
  Running the Program 1-8
Naming Conventions 1-8
  Variable Naming 1-8
    Prefixes 1-8
    Roots 1-10
    Suffixes 1-11
  Examples of Variable Names 1-11
  Procedure Naming 1-12
  File Suffix Conventions 1-12
Models of Computation 1-14
  Your Compiler and Models of Computation 1-14
Procedural Calls Between Programming Languages 1-15
  Your Compiler and Calls Between Languages 1-15

Some Configuration Tips for Programmers 1-16
Installing the Debugger 1-16
The :EnterDebuggerOnFault: Config.sys Parameter 1-16
Using Consistent Library Versions 1-16

2 Introduction to Protected Mode

Introduction 2-1
Advantages of Protected Mode 2-1
Review of Segmented Addressing 2-2
Review of Real Mode Addressing 2-2
Protected Mode Addressing 2-3
Memory Mapping Using 80386 Paging 2-6
The Format of a Selector 2-7
The Segment Descriptor Tables 2-7
Local Descriptor Table (LDT) 2-7
Global Descriptor Table (GDT) 2-8
The Contents of a Segment Descriptor 2-9
Processor Exceptions and Faults 2-11
Gate Descriptors 2-12
Protection Models 2-12

3 Using Color

Introduction 3-1
Video Concepts 3-2
Program Interfaces 3-3
ProgramColorMapper 3-4
ProgramColorMapper Structures 3-4
Single-Palette Format 3-5
Advantages Over Three-Palette Format 3-7
Three-Palette Format 3-8
Color Structure 3-9
Field Descriptions 3-10
Enabling Background Color Capability 3-12
Single-Palette Example 3-13
Three-Palette Example 3-14
Workstations with a 16-Entry Palette 3-15
More Sample Palettes 3-15
Case 1: Condition A 3-16
Case 1: Condition B 3-17

Case 1: Condition C 3-17
Case 1: Condition D 3-18
Case 2: Condition A 3-18
Case 2: Condition B 3-19
Case 2: Condition C 3-20
Gray-Scale Monitors 3-21
Application Notes 3-22
Combining Alphanumeric Color With the Values in the Attribute Byte 3-22
Defining Bitmap Color 3-24
Color Priorities 3-25
Programming Tips 3-28
Avoid Use of Reverse Video and Graphics 3-28
Obtaining Single-Palette Format 3-28
Avoid Combining Alpha Background with Graphics 3-28
Program Examples 3-29
Three-Palette Format Example 3-29
Single-Palette Format Example 3-36

4 Writing Partition-Managing Programs
A Review of Partition Management Operations 4-1
Creating a Partition and Loading a Program into It 4-2
Creating a Partition and Swapping It into Memory 4-2
Setting Up the Program's Environment 4-3
Loading the Program into the New Partition 4-5
Finding the Program's Termination Status 4-6
The Child-Termination Question 4-6
A Solution – Defining a Termination Request 4-7
Calling the Termination Procedure 4-8
Structure of the Termination Procedure 4-8
Deallocating the Partition 4-10
A Sample Partition Management Program 4-11

5 Software Installation: The Installation Manager

Introduction 5-1
Key Concepts 5-2
  Types of Installation: Floppy, Tape, Server 5-2
  Public vs. Private Installation 5-2
  Subpackages 5-3
The Installation Manager 5-3
Batch 5-5
Installation Files 5-5
  Control File 5-6
  Installation Script File 5-11
  Message File 5-11
  Command File 5-12
User Configuration File 5-13
Naming Your Floppy Installation Files 5-14
Naming Your Tape Installation Files 5-17
Organizing Your Installation Media 5-18
  For a Floppy Installation 5-18
  For a Tape Installation 5-20
Installation Variables 5-21
File Lists 5-24
Restarting an Installation 5-25
Nationalization 5-26
Tips 5-27
Example 1: One Subpackage 5-30
  Install.ctrl File 5-30
  InstallMsg.bin File 5-31
  Install.cmds File 5-32
  Install.jcl File 5-33
Example 2: Nested Subpackages 5-39
  Install.ctrl File 5-40
  DevelopmentUtilities>Install.ctrl File 5-41
  DevelopmentRunFiles>Install.ctrl File 5-42
  DevelopmentRunFiles>InstallMsg.bin File 5-42
  DevelopmentRunFiles>Install.cmds File 5-42
  DevelopmentRunFiles>Install.jcl File 5-43
  DevelopmentLibraries>Install.ctrl File 5-49
  DevelopmentLibraries>InstallMsg.bin File 5-50

DevelopmentLibraries>Install.jcl File 5-51
AsynchronousExamples>Install.ctrl File 5-57
AsynchronousExamples>InstallMsg.bin File 5-58
AsynchronousExamples>Install.cmds File 5-58
AsynchronousExamples>Install.jcl File 5-59
Example 3: Tape Installation 5-68
InstallMsg.bin File 5-69
Install.jcl File 5-70

6 Using the System Log File
Introduction 6-1
Log File Format 6-1
Writing Messages to the Log File 6-2
Displaying Messages Using PLog 6-2
Log File Fields in the Volume Home Block 6-3
How the File System Writes to the Log File 6-3
Record Fits in the Buffer 6-3
Record Does Not Fit in the Buffer 6-4
How to Read the Log File 6-5
Accessing the Log File Fields in the VHB 6-6
Determining if Records Have Wrapped Around 6-6
Reading the File in Chronological Order 6-6
If Records Have Not Wrapped Around 6-6
If Records Have Wrapped Around 6-7
Reading the File in Reverse Chronological Order 6-8
Saving Offsets to Records 6-8
If Records Have Not Wrapped Around 6-8
If Records Have Wrapped Around 6-8
PLog's Algorithm For Processing Each Sector 6-9

7 Writing System Services for the XE-530
Portation Issues for Existing Programs 7-1
General Guidelines 7-1
User Numbers and Exchanges 7-2
Remote Memory and Inter-CPU Communication 7-2
Inter-CPU Communication Buffer Block Size Issues 7-3
The Demise of the MCommands 7-3
Restriction on the GetWSUserName Operation 7-4

Controlling the Routing of Requests on the XE-530 7-4
The SRP Request Routing Directives 7-4
Local Routing 7-6
Remote Routing.................................. 7-7
Routing by Device Specification 7-10
Other Routing Methods 7-17
Use of Handles on the XE-530 7-18
The Standard Connection Handle 7-18
Non-Standard Handle Types 7-19

8 The Synchronous CommLine Interface

The CommLine Interface 8-1
Extensions to the Traditional CommLine Interface 8-1
InitCommLine 8-2
Differences Between 8274 and 82530 Communication Controllers 8-3
ChangeCommLineBaudRate 8-4
ReadCommLineStatus 8-4
WriteCommLineStatus 8-5
Using DMA with Synchronous Data Communication 8-5
Initializing Communications DMA 8-5
TransmitCommLineDMA 8-6
ReceiveCommLineDMA 8-7
GetCommLineDMAStatus........................... 8-8
Implementing X.21 (1984) Protocol Support 8-8
What Is the X.21 Protocol? 8-8
Features of the X.21 Support Hardware 8-9
Initializing a Communications Line with X.21 Support Enabled... 8-11
Using the X.21 Support Hardware in Drivers-Only Mode..... 8-12
Using V.35 Support Hardware 8-12

9 The SCSI Manager Target Mode

Introduction 9-1
SCSI Commands 9-2
INQUIRY 9-3
RECEIVE 9-5
REQUEST SENSE 9-6
SEND 9-9
SEND DIAGNOSTIC 9-10
TEST UNIT READY 9-11
SCSI Messages 9-12
ABORT 9-13
BUS DEVICE RESET 9-13
DISCONNECT 9-13
IDENTIFY 9-14
INITIATOR DETECTED ERROR 9-14
SYNCHRONOUS DATA TRANSFER REQUEST 9-15
Guidelines for SCSI Processor Target Mode 9-16
Single-Threaded Mode 9-16
Multi-Threaded Mode 9-17
Illegal Transfer Lengths 9-18
Finishing a Target Mode Application 9-19

10 Making CTOS Requests from MS-DOS

Why Call CTOS from DOS? 10-1
What Is CSKNAMES.OBJ? 10-1
Kernel Primitives Supported by CSKNAMES.OBJ 10-2
Using CSKNAMES.OBJ 10-3
The PC Emulator Version Port 10-4
A Sample Program Using CSKNAMES.OBJ 10-5

Index I-1

List of Figures

1-1. Where CTOS System Calls Are Processed 1-4

2-1. Real Mode Segment Addressing 2-3
2-2. Protected Mode Segment Addressing 2-5
2-3. Protected Mode Address Translation for Paging 2-6
2-4. The Format of a Selector 2-7
2-5. Pointer Aliasing 2-10
2-6. Separate Address Spaces Protection Model 2-13

3-1. Character Cell with Foreground and Background Pixels 3-2
3-2. Example Palette in the Single-Palette Format 3-6
3-3. Three-Palette Format and Single-Palette Format 3-11
3-4. Sample Single-Palette Format 3-13
3-5. Sample Three-Palette Format 3-14
3-6. Case 1 Palette Settings 3-16
3-7. Case 2 Palette Settings 3-19
3-8. Sample Palette 3-26
3-9. Color Priorities 3-26

6-1. Log File Wraparound 6-5

List of Tables

1-1. Common Prefixes 1-9
1-2. Common Roots 1-10

7-1. SRP Request Routing Types 7-5

8-1. 8274 and 82530 Register Differences 8-3
8-2. X.21 Status Indication Bit Patterns 8-10

Source Code Listings

3-1. ThreePalette.c 3-29
3-2. NewPalette.c 3-36

4-1. Creating a Partition 4-2
4-2. Initializing a New Partition's Environment 4-4
4-3. Loading a Task for Execution 4-6
4-4. Calling a Child-Termination Procedure 4-8
4-5. A Child-Termination Procedure 4-8
4-6. A Partition Removal Procedure 4-10
4-7. A Simple Partition-Managing Program 4-11
4-8. A Termination Request Definiton File 4-18

6-1. PLog's Record-Processing Algorithm 6-9

7-1. Sample Request.txt File Using Remote Routing 7-8
7-2. Sample Request.txt File Using Device Routing 7-11
7-3. A Client Program that Supports Device Routing 7-14
7-4. Changes to Sample System Service to Illustrate Device Routing 7-16

10-1. DOS Function Definitions for the Kernel Primitives 10-3
10-2. Determining PC Emulator Version From DOS 10-4
10-3. Calling a PC Emulator Version Procedure 10-5
10-4. A Sample DOS Program Using CSKNAMES.OBJ 10-6

About This Manual

This chapter provides a general road map for using this guide, and explains this guide's relationship to other CTOS documentation.

What Is the CTOS Programming Guide?

This Guide is a two-volume set, which describes many of the aspects of programming in the CTOS environment.

A separate book, the *CTOS/Open Programming Practices and Standards* guide, should also be considered part of this set. *CTOS/Open Programming Practices and Standards* describes the interfaces to features that are common to all CTOS-based operating systems. It also contains many sample programs, which illustrate how to use various aspects of the CTOS operating system. If your programs follow the guidelines in that guide, they should be both upwardly and downwardly compatible with most versions of CTOS and with most other CTOS-based operating systems.

Volume I of the *CTOS Programming Guide*, "General Programming Topics", explains the programming interfaces to many internal features of the operating system. It focuses primarily on features which may not be common to all versions of the operating system, or to all hardware platforms. For example, the chapter on shared resource processor programming clearly applies only to that type of hardware.

Volume II, "Extended System Services and Libraries", explains the programming interfaces to features which are not internal to the operating system, but which are packaged with it. For example, the Mouse System Service is a separate system service which a user can choose not to install. However, all users receive the Mouse System Service with their copy of the operating system.

What's New in This Update

In Volume I, Chapters 1 and 3 have been expanded with additional explanations and examples. Chapter 5 has been completely updated to reflect changes in the Installation Manager software.

In Volume II, Chapter 4 has been updated to reflect the addition of the Audio Service and the Series 5000 workstation. Changes made to Chapter 5 are mainly editorial in nature. Volume II also contains two new chapters: Chapter 7, "CD-ROM Service," and Chapter 8, "Sequential Access Service." A new index is included for both volumes.

Who Should Use This Guide

The *CTOS Programming Guide* is intended for programmers who want to write programs in the CTOS environment. Before using this guide, you should be familiar with CTOS as an end user and you should have some experience programming under CTOS or another operating system (preferably a multitasking one).

As mentioned above, the *CTOS Programming Guide* should be used in conjunction with *CTOS/Open Programming Practices and Standards*.

Structure of this Guide

Each section of this guide describes a different topic in CTOS programming. The sections are ordered with the features of more general interest toward the beginning, and the more esoteric or product-specific features toward the end.

To use this guide, read the sections on the features in which you are interested.

Related Documentation

The following manuals are part of the CTOS documentation set, and are likely to be of use to you. Additional programming tools and libraries are also available. Contact your sales representative for more information.

CTOS Documentation Directory
This quick reference card describes each of the manuals in the set and points the user to specific types of information.

CTOS Executive User's Guide
This procedural guide explains how to use the Executive command prompt and command forms. It also explains the file system and provides step-by-step procedures for performing common tasks, such as copying or deleting files, backing up to floppy disk, and initializing floppy disks.

CTOS Executive Reference
This reference manual is organized alphabetically by command name. It includes detailed information about the Standard Software commands and special features of the Executive.

CTOS Editor User's Guide
This guide describes how to use the new, enhanced Editor. It covers basic use of the Editor to create and modify an ASCII text file. It also describes new features added to the Editor to enhance it's use as a programmer's tool.

CTOS System Administration Guide
This guide contains general information about hardware types and system software products. It provides detailed information about installing system services, user configuration files, formatting disks, backing up data, optimizing performance, configuring and customizing operating systems, and troubleshooting common problems.

CTOS Development Utilities Reference Manual
This guide describes using the Linker, Librarian, and Assembler.

CTOS Debugger User's Guide
This guide describes how to use the Debugger commands to debug programs on real and protected mode operating systems. The manual provides hands-on exercises in using the commands as well as debugging tips.

CTOS Operating System Concepts Manual
This manual describes the BTOS II 3.2, CTOS/XE 3.0, CTOS/VM 2.4, and CTOS 9.10 (nine point ten) operating systems. It provides an explanation of how the operating system works and gives some orientation to the basic concepts the CTOS or BTOS programmer needs to understand.

CTOS/Open Programming Practices and Standards
This how-to guide describes the commonly-used, hardware independent aspects of programming under CTOS. It covers basic I/O, error handling, parameter management, guidelines for protected mode programming, writing nationalizable programs, writing system services, stack format and calling conventions, mixed-language programming, writing multiprocess programs, overlays, customized SAM, and communications programming. It includes programming examples.

CTOS Procedural Interface Reference Manual
This alphabetically organized reference manual covers each of the programming operations for BTOS II 3.2, CTOS/XE 3.0, CTOS/VM 2.4, and CTOS 9.10 (nine point ten). It also includes descriptions of certain operating system data structures.

CTOS Status Codes Reference Manual
This reference manual lists status codes returned by application programs and by the operating system. Codes are listed numerically. The second volume lists bootstrap errors.

CTOS Batch Manager II Installation, Configuration, and Programming Guide
This guide describes the use of the foreground and background Batch products.

CTOS Sort/Merge Programming Reference Manual
This guide describes Sort/Merge utilities and the Sort/Merge object module library.

CTOS Indexed Sequential Access Method (ISAM II) Programming Reference Manual
This guide describes programming with the ISAM libraries.

1

The CTOS Programming Environment

Introduction

This chapter gives a general overview of the CTOS environment as it relates to programmers. The chapter also identifies some specific issues that may apply to your environment, depending on which program development tools you use.

This chapter assumes that you are familiar with CTOS and with the Executive. If you are not, refer to the manuals listed in the "Related Documentation" section of this manual. If you are new to CTOS, you should pay particular attention to the *CTOS Executive User's Guide* and the *CTOS Operating System Concepts Manual*.

In addition, the *CTOS/Open Programming Practices and Standards* guide contains extensive programming documentation, with example programs, covering many of the most popular CTOS operating system features. It is also designed to serve as a CTOS primer for programmers.

Distributed Client-Server Computing: The CTOS Cluster

The CTOS cluster is a distributed computing environment based on the client-server model. A cluster consists of one server and one or more cluster workstations. When an application program executing on a cluster workstation calls the operating system, it sends a request to the operating system at that workstation.

If the local operating system is unable to provide the requested resource, it automatically forwards the request to the operating system at the server. The server then attempts to provide the requested resource. This process occurs transparently to the application program.

This transparent routing of operating system calls is the foundation of the distributed CTOS environment. It allows applications to take advantage of the CTOS cluster network without any special effort on the part of the application programmer. In effect, any application that runs on a CTOS workstation is a networked application.

For more detailed information about the request mechanism and about related CTOS internals, see the *CTOS Operating System Concepts Manual.*

Tools Available to the CTOS Programmer

An extensive array of program development tools are available under CTOS. The list below enumerates some of the most commonly used packages.

CTOS Development Utilities

The Development Utilities package includes an Assembler, Linker, Librarian and Debugger. It also includes *Ctos.lib* and *CtosToolKit.lib*, two libraries of object modules which provide the programming interface to the operating system. The Development Utilities are a prerequisite for programming under CTOS.

CTOS Developer's Tool Kit

The Developer's Tool Kit consists of over 40 utility programs of various types. Some are unique to CTOS and perform functions useful in the CTOS environment. Others are industry-standard utilities common on many platforms, such as the UNIX utilities Make, Yacc and Awk.

Assorted Language Compilers and Interpreters

In addition to the utility packages, compilers and/or interpreters are available for most popular computer languages, including Basic, C, Cobol, Fortran, Pascal, and PL/M.

Additional Programmer's Libraries

The CTOS Development Utilities contain the essential libraries, but additional libraries are available, both from Unisys and from third-party developers. These libraries provide support for such

features as device-independent printing, display of graphics, and interfaces to electronic mail, among others.

Take the time to familiarize yourself with the tools at your disposal, and with the other tools that may be available, before you plunge into programming under CTOS.

An Overview of CTOS System Software

CTOS system calls consist of four types of entity: kernel primitives, requests, system common procedures, and object module procedures. The description of each procedure in the *CTOS Procedural Interface Reference Manual* specifies which type it is.

A kernel primitive is a direct command to the CTOS kernel. Kernel primitives can be used when a lower-level interface to the operating system is needed. Most programs use the kernel primitives only indirectly, by using the procedural interface for requests.

A request passes through the CTOS kernel, which determines where to route the request for processing. Most internal operating system features are implemented using requests. There is also a special class of application program, called a system service, which serves application-defined requests.

A system-common procedure is a subroutine which is part of the operating system and can be called directly. Calls to system common procedures do not pass through the CTOS kernel. Instead, CTOS uses a feature of the Intel microprocessors (a Call Gate) to redirect the call to the appropriate subroutine within the operating system. As with requests, system service programs can install their own system common procedures for use by other applications.

An object module procedure is simply a subroutine stored in *Ctos.lib* or in *CtosToolKit.lib*. If your program calls one of these procedures, the Linker links the object module that contains the procedure to your program at link time. That procedure then becomes part of your program.

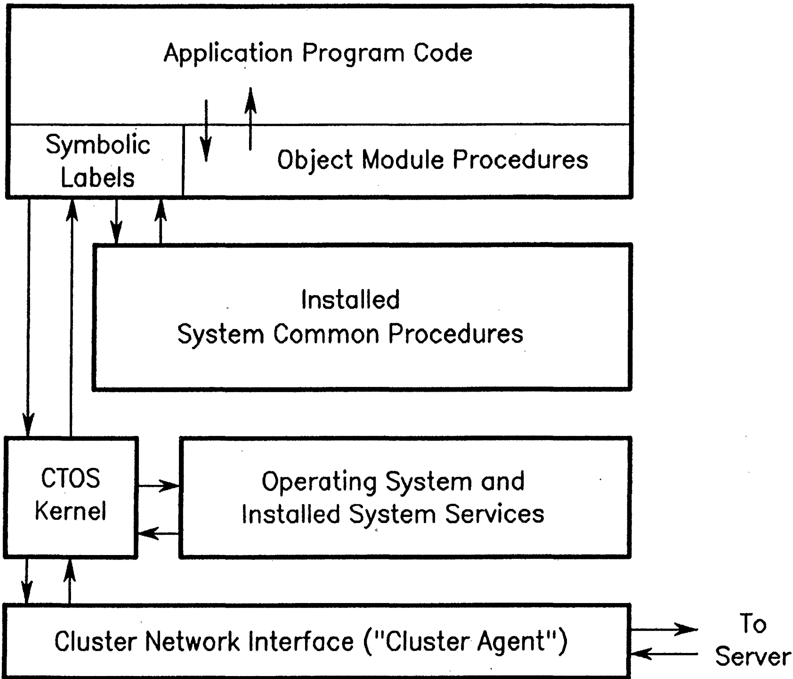


Figure 1-1. Where CTOS System Calls Are Processed

Figure 1-1 shows the level at which the different types of procedures execute, relative to the CTOS kernel. As you can see from the figure, the kernel controls access to the cluster network. Therefore, only those system calls that pass through the kernel can be routed across the cluster.

Starting at the top, observe that object module procedures actually become part of your program when you link it. Calls to these procedures remain "inside" your program. Their code is linked into your program, and your program's thread of execution executes the code.

Shown below the object module procedures are the installed system common procedures. These procedures are not part of your program, but they do execute on the local processor. Your program's thread of

execution executes the code in a system-common procedure, but the code itself is not linked into your program.

On the third level is the CTOS kernel. All requests your program makes are routed through the kernel, which forwards them to the local operating system or to an installed system service (if the appropriate one is available). Otherwise, the kernel sends the request over the cluster network to the server for processing.

General Steps in Writing a CTOS Program

The following topics describe the general process of writing and running a program under CTOS. This section assumes you are using a compiled language, such as C. If you are using an interpreted language, such as Basic or COBOL, the steps may be somewhat different. See your language interpreter manual for more information.

Writing Source Code

Several high-level languages and assembly language are supported. You can write programs entirely in these languages, if they provide all the functionality you need. Your program can also call CTOS operations directly. See your compiler documentation for more information.

Most CTOS programmers use the Editor to write their source code. You can, however, use other text processing programs, such as Document Designer or OFISDesigner. The Editor, Version 11.0 or higher, provides many of the features of a text processor, such as opening multiple files in different windows. In addition, the Editor includes other features designed specifically for the programmer, such as block checking. For details on the Editor, see the *CTOS Editor User's Guide*.

In most languages, programs are written in *modules*, or subparts, which are linked together later into a complete program. You create a separate source code file for each module.

The CTOS system naming convention is to attach a period and a suffix to the name of a source file. The suffix indicates the language in which the source code is written, for example,

MyProgram.asm	assembly language
MoreCode.pas	Pascal
StringOps.c	C

Usually, you should give your source code text file the same name you will use for the compiled, object code version of it. You should also use this file name if your programming language requires you to explicitly name the module inside the source file. Following this convention makes the Linker's map file much easier to read, and helps you keep track of your source code.

Compiling

After you have written the code, the next step is to compile or assemble the modules of your program. See your compiler documentation for any special requirements.

Most compilers generate the following two output files:

- A *list file* from the compilation. This file is automatically named. Your source file name suffix (if any) is dropped, and the suffix *.lst* is appended to the root name to create the list file name. For example, the list file for *StringOps.c* is named *StringOps.lst*.

  The compiler writes any compilation errors it detects to this file. You can examine the file by using the **Type** command and specifying the list file name in the command form. You can also print it out or edit it.

- An *object file*, if the compilation was successful. The object file contains the executable code generated by the compiler. The name of this file is the root file name with the new suffix *.obj*.

Linking

An object file is not a run file. You cannot execute an object file directly: it must be linked first, even if there is only one object module in your program.

To make your program executable, start the Linker and specify the names of the object modules you want linked along with the name to give your *run file*. This name, unlike those from the compiler, is not assigned automatically. It is useful to give this file name the suffix *.run*.

The **Link V6** command starts the Linker and tells it to create a program that can run in protected mode under CTOS II, but can still run in real mode on older processors. When you use the **Link V6** command, enter the keyword *Protected* in the command form field, *[Run file mode]*. To create a real-mode-only program, which runs in real mode no matter which CTOS operating system is present, you can specify *Real* as the run file mode.

When you use either Linker command, you also may need to specify other parameters, such as the names of libraries that you want searched. For details on using the Linker, see the following sources:

- The *CTOS Development Utilities Reference Manual.*
- "Protected Mode Programming Guidelines" in the *CTOS/Open Programming Practices and Standards* guide.
- The section on linking programs in your language manual.

If the Link is successful, three more files are generated:

- The *run* file for your program.
- A *symbol* file, which is named *RunFileName.sym*. The symbol file is used during debugging. It maps the symbolic names you gave your variables to their locations in your program's image in memory. This lets you look up the value of MyVar in the Debugger, instead of having to look up the value of an address, such as DS:04E2h.

- A *map file,* which shows the relative address in the memory image and the length of each segment of the program, as well as any errors encountered during the link. The map file has the name *RunFileName.map*. You can examine it by using the **Type** command.

Running the Program

Use the **Run** command to execute your run file. **Run** is a generic command for executing files that do not have Executive commands of their own. The **Run** command simply presents a numbered list of parameter fields, with no special labels to identify runtime parameters.

To give users a conventional means of executing the run file, create a new Executive command for your program using the **Command File Editor** or the **New Command** command.

Naming Conventions

CTOS programmers are encouraged to use the CTOS naming conventions for variables, procedures, and files.

Variable Naming

The name of the variable implies some of its characteristics. Parameters used in procedure definitions and fields of request blocks and other data structures are named according to this convention.

A variable name is composed of up to three parts: a prefix, a root, and a suffix.

Prefixes

The prefix identifies the data type of the variable. Common prefixes and the number of bytes required for each are shown in Table 1-1. Note that a flag passed to the operating system or returned by an operating system procedure is interpreted as meaning TRUE if its value is 0FFh and FALSE if its value is 0. Even if you are using a high-level language that uses a different value for TRUE or FALSE, you must use these values to mean TRUE and FALSE in operating system calls.

Table 1-1. Common Prefixes

Prefix	Bytes Required	Description
b	1	Byte (character or unsigned integer).
c	2	Count (unsigned integer).
cb	2	Count of bytes (in a string of bytes).
f	1	Flag (TRUE = 0FFh and FALSE = 0).
i	2	Index (unsigned integer).
l	varies	literal (a constant).
n	2	Number (unsigned integer) (same as c).
o	2	Offset from the segment base address.
p	4	Logical memory address (pointer) consisting of a segment address and an offset.
pb	4	Logical memory address of a string of bytes.
q	4	Quad (unsigned integer).
rg	varies	Array. Usually used with another prefix, for example, the prefix *rgb* identifies an array of bytes.
mp	varies	Map. A one-to-one correspondence between two variables. Usually used with other variables being mapped, for example, *mpcParcRq* maps the count of paragraphs to the count of requests of that size
s	2	Size in bytes (unsigned integer).
sb	varies	String. An array of bytes where first byte is the size of the string.
sz	varies	String. An array of bytes with a null terminating byte.
w	2	word

Roots

The root can be a unique name for a variable (such as *clientA*), it can be a general name such as the examples shown in Table 1-2, or it can be a combination of a unique name and a general name (such as *exchClientA*). Common roots and the number of bytes required for each are shown in Table 1-2.

Table 1-2. Common Roots

Root	Bytes Required	Description
erc	2	Error code.
exch	2	Exchange.
fh	2	File handle.
fhb	*	File Header Block (FHB). (See "System Structures" in the *CTOS Procedural Interface Reference Manual* for the size of the FHB system structure.)
lfa	4	Logical file address.
qeh	4	Queue entry handle.
rq	varies	Request block. Size varies with the request.
sa	2	Segment address (high-order two bytes of a logical memory address) An sa may be either an sn or an sr. (See below.)
sg	2	Global Descriptor Table (GDT) selector.
sl	2	Local Descriptor Table (LDT) selector.
sn	2	Selector (high-order two bytes of a protected mode logical memory address).
sr	2	Paragraph number (high-order two bytes of a real mode logical memory address).
userNum	2	User number.

Suffixes

The suffix identifies the use of the variable. Suffixes are

Last	Largest allowable index of an array.
Max	Maximum length of an array or buffer (thus one greater than the largest allowable index).
Ret	Identifies a variable whose value is set by the called process or procedure rather than by the caller.

Examples of Variable Names

cbFileSpec	Count of bytes in a file specification.
ercRet	Error code to be returned to the caller.
pbFileSpec	Memory address of a string of bytes containing a file specification.
pDataRet	Memory address of an area to which data is to be returned to the caller.
ppDataRet	Memory address of a 4-byte memory area to which the memory address of a data item is returned to the caller.
pRq	Memory address of a request block.
sData	Size (in bytes) of a data area.
sDataMax	Maximum size (in bytes) of a data area.
psDataRet	Memory address of a 2-byte memory area to which the size of a data item is returned.
ssDataRet	Size of the memory area to which the size of a data item is returned.

Procedure Naming

The name of a public procedure (one called by modules other than the one where it is defined) should describe its function and should be specific enough so that it does not conflict with other procedure names. It is a convention to begin each new word or abbreviation within a procedure name with a capital letter:

GetDateTime
PosFrameCursor
QueryDefaultRespExch

Procedures that are part of a system service usually consistently contain an abbreviation that denotes that system service:

TsConnect
TsDial
TsEnableRing
CloseISAM
DeleteISAM
GetISAMRecords

File Suffix Conventions

The type of a file is denoted by the suffix on its name, as was noted earlier in this section. Some of these suffixes are required, and some are optional. For those related to programming, compiling, and linking, see the appropriate language manual for more information. Whether required or not, it is a good idea to use the suffixes because they let you determine a file's use from its name.

Here is a summary of some common suffixes. These are related to programming and to system files. A few other suffixes exist in file names generated by various application programs.

Suffix	Use
.asm	assembly language source file
.awk	Awk script file
.bas	BASIC source file

Suffix	Use
.bin	a binary message file
.c	C language source file
.cbl	a COBOL source file
.config	program configuration file
.edf	external definition file
.fls	a file containing a list of files: usually an at-file
.for	FORTRAN source file
.form	object module created by Forms Editor
.gnt	a COBOL intermediate-format file
.h	C-language header file
.idf	internal definition file
.img	bootable protected mode image
.int	a COBOL intermediate-format file
.jcl	a job-control text file
.lib	a library file created by Librarian
.lst	list file from compiler
.map	map file from Linker
.mdf	assembly language macro definition file (may contain other definitions also)
.obj	object module from compiler or assembler
.pas	Pascal source file
.plm	PL/M source file

Suffix	Use
.run	run file from Linker (not assigned by default)
.sed	Sed script file
.sub	submit file, read by **Submit** command
.sym	binary symbol file from Linker, defining public labels for use by Debugger
.sys	system file, used by operating system
.tmp	temporary file, created by standard software or application program
.txt	a message text file
.user	user configuration file

Models of Computation

The Intel architecture uses a number of "models" of computation. These models are based on the way a program is divided. For example, in the small model, all the program's code and data is lumped together in a single segment.

CTOS uses the medium model, in which each object module has its own separate code segment. However, all the global data for the program, plus its stack, are placed together in a single data segment called DGROUP.

Your Compiler and Models of Computation

The language compilers and interpreters available from Unisys and from third-party developers use widely varying models of computation. Many also support more than one model.

The model you use becomes significant when your program needs to make an operating system call, because the model your compiler uses may differ from the one CTOS uses. Compilers and interpreters that cannot use the

medium model generally provide an interface mediator for operating system calls. See your compiler documentation for more information about how to make operating system calls from that language.

For more general information about models of computation, see "Stack Format and Calling Conventions" in the *CTOS/Open Programming Practices and Standards* guide.

Procedural Calls Between Programming Languages

Calls to the operating system can be thought of as calls to a different programming language. You may also find yourself in a situation where you need to call C routines from Cobol, or something similar.

There are several issues involved when you mix programming languages, but the two primary ones are model of computation and stack format.

The language compilers and interpreters available from Unisys and from third-party developers generally provide a simple mechanism that allows you to make operating system calls from that language. See your compiler documentation for more information.

For more general information about procedural calls between programming languages, see "Mixed-Language Programming" in the *CTOS/Open Programming Practices and Standards* guide.

Your Compiler and Calls Between Languages

When you need to call non-operating system procedures written in a different programming language, the best strategy is to map both to the CTOS calling convention. The CTOS calling convention is described in "Stack Format and Calling Conventions" in the *CTOS/Open Programming Practices and Standards* guide.

If the language you need to call can use the CTOS calling convention natively, then your job is easy. Use the CTOS calling convention for the procedures you need to call, and have the calling language pretend that they are operating system procedures.

If the language used for the called procedures cannot use the CTOS calling convention, the task is more difficult. You need to carefully

review the documentation for each compiler, and try to identify a calling convention that both can use.

Some Configuration Tips for Programmers

This section collects some miscellaneous information that can help you set up your development system.

Installing the Debugger

All you need to do to install the CTOS debugger is install the files for it. If the Debugger file resides in your boot directory (where sysimage.sys resides) the operating system loads the Debugger into memory automatically.

See the *CTOS Debugger User's Guide* for the name of the Debugger file used by your operating system.

The :EnterDebuggerOnFault: Config.sys Parameter

When developing programs that run in protected mode, bugs often manifest themselves as protection violations. To trap these violations when they occur, add the following line to your *Config.sys* file:

```
:EnterDebuggerOnFault:Yes
```

This parameter causes the operating system to enter the Debugger whenever it detects a protection violation. You then have the opportunity to trace the cause of the fault before your program exits.

Using Consistent Library Versions

When you update your development environment, take care to maintain its consistency. If you developed a program using one version of *Ctos.lib*, you should keep that version until you want to update the program. Using incompatible versions of software can cause perplexing errors.

One such pitfall is that the contents of *Ctos.lib* changes from release to release as features are added. The CTOS developers go to great pains to

prevent incompatibilities, but they do occur. For example, a new operating system call could have the same symbolic name as one of the procedures in your program. If the Linker encounters such a situation, it returns an error message.

A second common pitfall is accidentally using a symbolic name that is already used in *Ctos.lib*. For example, a program might define a global variable, *fProtectedMode*. This name also corresponds to a procedure in *Ctos.lib*. Such a program will link successfully, then cause a protection fault when it executes.

A third potential pitfall is that some third-party object module library you use could depend on a particular version of *Ctos.lib*. If this type of problem occurs, your program may link successfully, then act strangely when you try to run it.

Finally, when you update your development environment, make sure you install all parts of your new development package. If you install a new version of *Ctos.lib* without new versions of its associated header ("EDF") files, you are likely to run into trouble.

As a general rule, you should keep close track of the configuration of your development environment. You should also save a copy of your previous library versions when you install new ones.

2

Introduction to Protected Mode

Introduction

This section is intended to familiarize you with protected mode concepts, so that you will understand how to write programs that run in protected mode on CTOS protected mode operating systems, such as CTOS/VM, CTOS/XE and BTOS II.

This section introduces you to features that are provided by Intel's protected mode microprocessors and are used by protected mode CTOS operating systems. For details on Intel microprocessor architecture, see the following Intel manuals:

- *iAPX 286 Programmer's Reference Manual*
- *80386 Programmer's Reference Manual*

In addition, this section compares real mode addressing to protected mode addressing. Understanding addressing differences can help you avoid compatibility problems.

To be sure that your programs are compatible (can be loaded and executed in either real or protected mode), you should follow the guidelines in "Protected Mode Programming Guidelines" in the *CTOS/Open Programming Practices and Standards* manual.

Advantages of Protected Mode

Protected mode offers advantages over real mode, including the following:

- A program can access a much larger address space (memory). The 1 megabyte limit of real mode does not apply.

- The hardware protection mechanism prevents a program from accidentally overwriting code or writing beyond the end of a segment.

Review of Segmented Addressing

On Intel microprocessors, instructions do not accept physical addresses as operands; they accept only segmented logical addresses. A *logical address* is formed from two 16-bit parts, the *segment address* (SA) and the *relative address* (RA). The RA is often referred to as an *offset* from the SA. The processor combines these two parts to identify a location in memory. When using the Assembler or Debugger, logical addresses are written as follows:

SA:RA

As each instruction executes, the processor forms a *linear address* from each logical address. The processor then uses the newly-formed linear address to access physical memory. Note that a program never uses a linear address directly. Programs always reference memory by using logical addresses.

The Intel architecture is referred to as a *segmented addressing model*, because of its two-part logical addresses. Every logical address is composed of a relative offset from some segment address (SA). Note the contrast to a *linear addressing model* such as the Motorola architecture, where instructions accept 32-bit linear addresses rather than SA:RA pairs.

When a logical address contains a *real mode* segment address (SA), that SA is sometimes referred to as an *SR*, to distinguish it from a *protected mode* segment address, which is sometimes called an *SN*. The abbreviation SA applies to either mode. Thus, in real mode, all logical addresses are actually SR:RA addresses. In protected mode, they are actually SN:RA addresses. The difference between an SR and an SN is described next.

Review of Real Mode Addressing

Real address mode is the only mode in which 8086 and 80186 microprocessors operate. Protected mode microprocessors, such as the 80286 and 80386, execute in real mode initially upon power-up or reset, but then switch to protected mode under the control of the operating system.

In real mode, to compute a 20-bit linear address result, the processor shifts the segment address left by four binary places (effectively multiplying it by 16), then adds the relative address.

The resulting 20-bit quantity can address only 1 megabyte of memory (2^{20} locations). For example, the linear address generated from logical address 3A02:235h is shown in Figure 2-1. Note that one hexadecimal digit in the figure corresponds to four binary address bits.

```
real address 3A02:235

 3A020  SR
+0235   RA
------
 3A255  20-bit linear address
```

2392.2-1

Figure 2-1. Real Mode Segment Addressing

In real mode, each segment address identifies a location 16 bytes higher in memory than the previous segment address. These 16-byte units of memory, each aligned on a 16-byte boundary, are called *paragraphs*. Therefore, a real mode segment address can be considered a *paragraph number*, because it denotes a particular 16-byte boundary in the physical address space.

In real mode, the *segment registers* contain the paragraph numbers corresponding to the base of the respective segments. The segments are always aligned to start on 16-byte boundaries.

Protected Mode Addressing

Protected mode microprocessors can address extended memory beyond the first megabyte. The amount of addressable memory is determined by the microprocessor and hardware limits. The 80286 microprocessor, for example, can address up to 16 megabytes of memory in protected mode. To do so requires a linear address of 24 bits. However, no Intel microprocessor, not even the 80286 or 80386, can address more than 1

megabyte in real mode. This is because only a 20-bit address can be formed by the address calculation described in the previous section.

To achieve this longer address in a compatible manner and to allow other features of protected mode to be implemented, the same two-part addressing scheme (the segmented addressing model) is used. The address still consists of an SA:RA, but the SA part of the logical address is interpreted differently. The RA part has the same meaning as in real mode.

In protected mode the 16-bit SA in the segment register is no longer a paragraph number, as it is in real mode. Rather, the SA is an index into a *descriptor table.* The SA is called a *selector* in protected mode because of the indexing or selection function it performs and is written as SN to distinguish it from a real mode segment address.

The operating system constructs and maintains a descriptor table called a *Local Descriptor Table* (LDT) for each run file executing in protected mode from information provided by the Linker in the run file header. Note that to produce the proper run file format, you must enter the keyword *Protected* in the Linker **Bind** command field, [Run file mode].

Each code and data segment in the program has a unique selector assigned by the Linker and a corresponding unique LDT entry. The LDT is an array of these entries, called *descriptors,* which contain various information about the segment. The selector is basically an offset into the LDT.

When an instruction loads a selector into a segment register in a protected mode program, the processor uses the selector to find the corresponding descriptor and to retrieve a segment base address from it. On the 80286 microprocessor, this base address is 24 bits long. By comparison, the base address is 32 bits long on the 80386 microprocessor, which has comparatively greater address space (up to 4 gigabytes). When an instruction refers to a memory address (using the segment register and an RA), the RA is added to the base address to obtain a linear address. Unlike real mode, no shift of this base address is done; it is not a paragraph number, but a true physical address.

Figure 2-2 shows this process schematically. Note that the example, 18:235, addresses the same memory location as the real mode example shown in Figure 2-1.

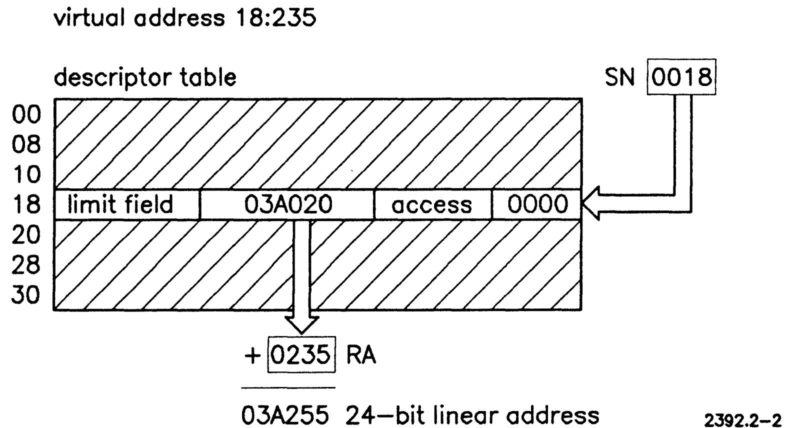


Figure 2-2. Protected Mode Segment Addressing

Figure 2-2 shows the descriptor holding the linear base address, 03A020. Instead of the segment register holding the paragraph number 03A02, it holds a selector value, 18, which is used as an index into the descriptor table.

The protected mode SN:RA logical address is sometimes referred to as a *virtual address*, because the SN refers only indirectly to memory via a descriptor table entry. By changing the descriptor table entry, the operating system can make the same virtual address refer to a different linear address (for example, to relocate the segment in memory without the program's knowledge).

The real mode SR:RA logical address is referred to as a *real address*, because it always corresponds to the same linear memory address.

Most incompatibilities between real mode and protected mode arise from this difference between paragraph numbers (SRs) and selectors (SNs). In particular, they arise most often from the assumption that a segment address is a paragraph number.

Memory Mapping Using 80386 Paging

The *physical address* is the actual location in system memory. In real mode and in protected mode without paging, there is a direct correspondence between the linear address and the physical address. When paging is enabled on 80386-based systems in protected mode, however, the linear address is mapped to the physical address by means of a page translation table. The operating system controls the use of this table in a manner transparent to programs. Figure 2-3 illustrates how addresses are mapped.

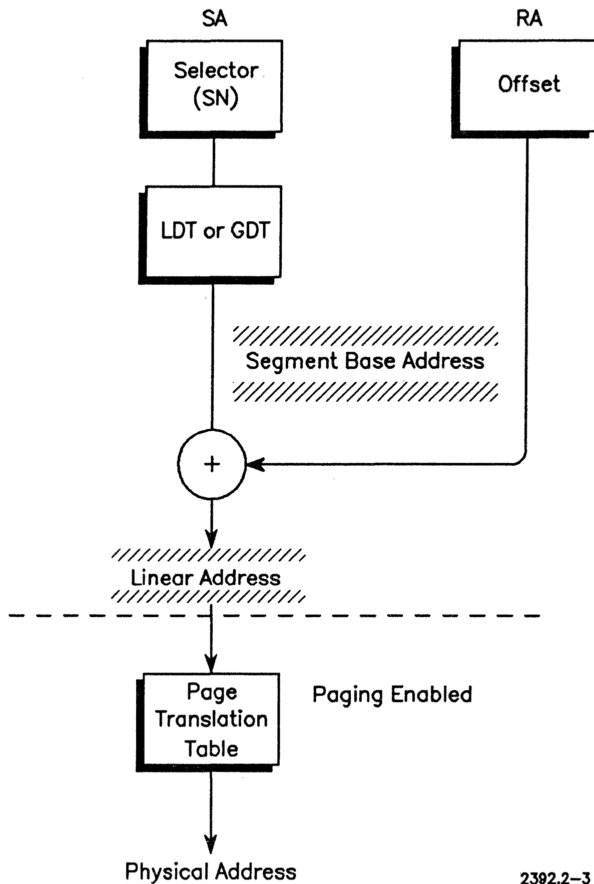
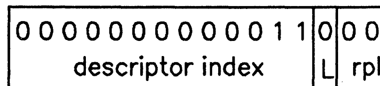


Figure 2-3. Protected Mode Address Translation for Paging

The Format of a Selector

The format of the 16-bit selector (SN) is shown in Figure 2-4. The high-order 13 bits form the offset that indexes into the descriptor table. The next bit discriminates between two kinds of descriptor tables, the Local Descriptor Table (LDT) and the Global Descriptor Table (GDT). The low-order 2 bits (indicated as *rpl* in Figure 2-4) currently are not currently used by CTOS.

SN 0018



L = 1 for LDT
L = 0 for GDT

2392.2-4

Figure 2-4. The Format of a Selector

The 13 index bits of the selector can generate 2^{13} or 8192 possible values. Thus the hardware permits 8192 entries in each of the two tables.

The Segment Descriptor Tables

Two kinds of descriptor tables are used to address segments: Local Descriptor Tables (LDTs), which were discussed previously, and the *Global Descriptor Table* (GDT). Typically, there is a separate LDT for each run file executing in protected mode. There is only one GDT per processor.

Local Descriptor Table (LDT)

At any one time, there is only one LDT in effect. This LDT is the *current LDT*. It describes the current program's code and data, and the entry

points (or call gates) at which the program can make system calls. (See "Gate Descriptors," later in this section.)

Each protected mode program may be associated permanently with an LDT when it is created. All processes of a program share the same LDT. When process switching occurs, the current LDT is automatically changed by firmware.

Global Descriptor Table (GDT)

The GDT describes *global* segments (segments that may be addressed by any program or the operating system). The operating system controls the use of the GDT. Currently, CTOS uses GDT-based segments for the following:

- operating system code and data
- dynamically allocated system data, such as character maps and the Application System Control Block
- code and data of programs that contain system-common procedures
- alias selectors for interprocess communication

Because operating system code and data are GDT-based, the operating system can be called at certain entry points at any time as a subroutine of the user process. The operating system services known as *Kernel primitives* and *system-common procedures* are implemented in this manner.

In addition, programs, such as the Video Access Method (VAM), that include system-common procedures must have GDT-based selectors when linked. (Such programs must be linked using the keyword *GdtProtected* in the Linker's **Bind** command form. See the *CTOS Development Utilities Reference Manual* for more information.) A protection mechanism known as a *call gate* limits the user program to calling at legitimate entry points and provides a convenient way to bind the user program to those entry points at program load time. (See "Gate Descriptors," later in this section.)

When an LDT-based program issues a request to a system service, the operating system gains control and changes the LDT-based pointers in the request block into GDT-based pointers. These GDT-based pointers are

called *alias pointers*. *Aliasing* allows the client and the system service to communicate with each other. The system service cannot directly use the client's LDT-based pointer, because the system service also may have an LDT that is different from the client's. Only one LDT can be current at a time.

Figure 2-5 on page 2–10 illustrates segment aliasing. Client A issues a request to System Service B. One of the request block parameters is a buffer accessible to Client A at address 84:0. The operating system gains control and creates an alias selector that also points to Client A's buffer. System Service B can access Client A's buffer at the GDT-based address, 100:0.

The Contents of a Segment Descriptor

Segment descriptors contain information about individual memory segments. In addition to the base address described in "Protected Mode Addressing," earlier in this section, each segment descriptor contains other information, such as

- the segment's size, which is indicated by its base and limit
- the segment type, such as code or data
- whether the segment is in memory or is swapped to disk

While a program executes in protected mode, the hardware performs various checks using information in the segment descriptor. These checks provide the following protection features:

- It is not possible to access memory beyond the end of the segment, either accidentally or deliberately.
- Data segments can be designated as writable or readable but can never be used as code segments.
- Code segments are implicitly executable but cannot be overwritten.

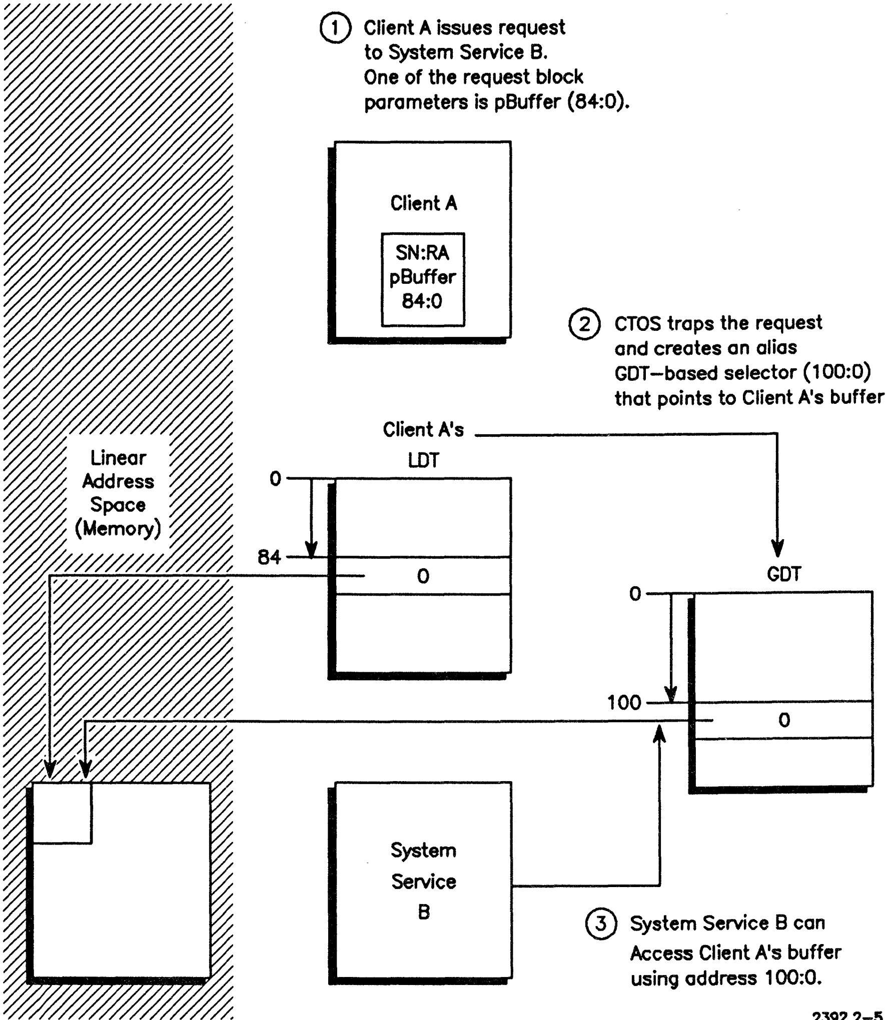


Figure 2-5. Pointer Aliasing

Processor Exceptions and Faults

A *fault* occurs if a selector is invalid or is used improperly. Faults transfer control automatically to the operating system.

Faults are of two types: *restartable faults* and *exceptions*. Restartable faults are, in theory, recoverable. Exceptions, on the other hand, are errors that prevent further execution of the program.

The *not-present fault* is an example of a restartable fault. Such a fault occurs when the operating system has marked a segment's descriptor to indicate that it is not currently resident, but is swapped out on disk. This kind of fault gives the operating system the opportunity to read the missing segment into any available free memory, fix up the base address in the descriptor, mark the descriptor *present*, and return to the interrupted program. The program then proceeds to try the instruction that faulted again, since the processor's instruction pointer still points to that instruction.

The use of an invalid selector because of a programming error is another example of an exception, called a *general protection fault*.

For instance, some real-mode programs use the ES register to store data. In protected mode, however, you can load only selectors into the ES register. If a program loads into ES any value that is not a selector, even before it attempts to use that value, the hardware's attempt to fetch the associated descriptor results in a fault.

There is one exception to this rule. The selector value 0 is special. It can be loaded into a segment register, but it causes a fault if it is subsequently used in an address calculation. The 0 value is permitted to allow the passing of NULL pointers, for example *pbPassword = 0* and *cbPassword = 0* in a CTOS OpenFile call. Outlawing NULL pointers would be extremely inconvenient, so they are allowed, but the value can never be used in calculating an address.

There are additional exceptions, which can occur later when a valid segment register is used in an address calculation. As an example, a *limit exception* occurs when attempting to address beyond the end of a segment.

For more detailed information about faults and exceptions, see the Intel documentation.

Gate Descriptors

A *gate descriptor* is a structure that allows a program to call routines, the addresses of which cannot be known until the program is loaded. Generally, any address that lies outside the run file will be unknown until load time. For example, a program may call a *system-common procedure* or a *Kernel primitive*, the location of which the Linker cannot supply.

To resolve such a reference, the Linker uses a *call gate* in the protected mode run file. At load time, the operating system fills the gate with the virtual address of the called procedure. The actual CALL instructions in the calling code segment, however, are not modified.

In the program code, these calls simply appear to call a virtual address using an ordinary far CALL instruction. When the call is executed, the processor uses the selector to fetch its associated descriptor. Upon examination of the descriptor, however, the processor determines that it identifies a call gate. Therefore, its destination fields are used in place of the original address to reach the appropriate routine. The call gate SN (not the original SN) is placed in the CS register. The original RA is ignored, and a value of zero is placed in the instruction pointer. This address is the first instruction of the called procedure.

In addition to the call gates just described, there are other types of gates with varying functions. *Task State Segment* (TSS) gates normally are used by the operating system. TSS gates also are involved in interrupt processing, as are two other gates: *interrupt gates* and *trap gates*. For more information on the function and use of these gates, see the Intel documentation.

Protection Models

Protected mode on Intel microprocessors actually uses two models of protection. Only one of these models, however, currently is used by CTOS. These models are summarized below:

- Protection by *separate address spaces*, shown in Figure 2-6, in which a program is restricted to one virtual address space and cannot even describe locations in another address space. CTOS uses this protection model. It provides isolation of one run file from another.

- Protection by *privilege level,* in which every program executes at one of several levels of authority. This protection model currently is not used by CTOS.

In Figure 2-6, each pie slice represents a separate address space, described by a separate LDT (the first protection model). The domain of each descriptor table is also shown.

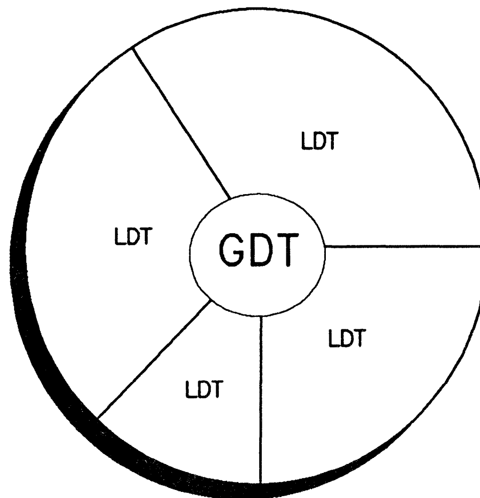


Figure 2-6. Separate Address Spaces Protection Model

A program in one address space is aware of and can describe only those locations for which it can load selectors. At any given time only one LDT is in effect (the LDT associated with the current process). The GDT is also in effect (all the time). Therefore, each process can address only those objects for which there exist descriptors in its own LDT or the GDT.

The LDT describes the program's own code and data, and the gates it may use to make system calls. In other words, it describes only those objects that it is legitimate for the program to access. The uses of the GDT are described in "Descriptor Tables," above.

3

Using Color

Introduction

This section describes how to use the ProgramColorMapper operation to define color for use in graphics and alphanumeric displays. ProgramColorMapper is an object module procedure available in the standard operating system library. Although the single-palette format provided by ProgramColorMapper (described in detail in this section) is available with VAM, Version 3.1 and later, the ProgramColorMapper operation itself is backward and forward compatible with all supported versions of the operating system and with Context Manager.

This section begins by discussing a few video concepts you need to understand when manipulating color. The section continues with an overview of ProgramColorMapper and other related color operations that call ProgramColorMapper. Following this introduction, the section describes the structures for setting up and controlling the way color will be displayed. At the conclusion of this section, there are two program examples, which show how to use ProgramColorMapper to set up palettes in either of two formats.

Before reading this section, you should familiarize yourself with the video concepts described in "Video" in the *CTOS Operating System Concepts Manual*. In addition, you should know how to assign attributes to characters and display characters to the video device using the VAM and VDM operations. If you are not familiar with these operations, review their detailed descriptions in the *CTOS Procedural Interface Reference Manual*. Finally, if you intend to write graphics applications, you should be familiar with the graphics library procedures. See the *Graphics Programmer's Guide* for details. Background on advanced color theory is left to the reader.

Video Concepts

Brief definitions of video concepts discussed in this section are presented below:

pixel — An illuminated dot displayed on the video device as a result of manipulating the alphanumeric character map or the graphics multi-plane bit map.

character cell — A rectangular pattern of pixels displayed to the video device. The pixels define a single alphanumeric character and the area surrounding that character. Depending on the hardware, character cells are displayed in different sizes (height and width in pixels) and in different resolutions (proximity of pixels in the character cell).

foreground color — Color displayed by the pixels in a character cell that define the character.

background color — Color displayed by the pixels in a character cell that define the area surrounding the character.

Imagine that the dots shown in Figure 3-1 are two colors: the filled-in dots represent one color, and the empty dots, a different color. All the dots represent pixels defining a character cell. The filled-in dots define the foreground alphanumeric color within the character cell. The empty dots define the background color. Some hardware (for example, workstations with Enhanced Video or VGA) can display multiple background colors. On such hardware, character cells adjacent to each other could display the background in different colors.

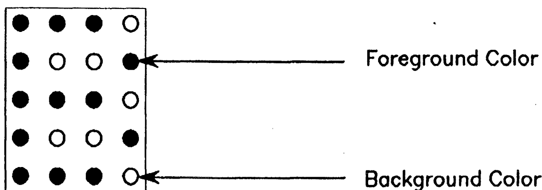


Figure 3-1. Character Cell with Foreground and Background Pixels

The concepts of foreground and background need to be taken into account when displaying alphanumeric information through VAM or VDM operations that manipulate the character map (on either character map workstations or on bit map workstations with character map emulation). When calling graphics routines to manipulate the multi-plane bit map, however, pixels are not viewed with regard to foreground or background because there is no character cell concept.

Using the single-palette format through ProgramColorMapper as described in this section, it is possible to use the same palette entry for graphics and alphanumerics. Manipulating color at the level of the pixel requires a priority to determine which pixel color will be displayed. This subject will be addressed in "Application Notes," later in this section after you have been introduced to the ProgramColorMapper structures.

Program Interfaces

To customize the use of color and to program the graphics control registers, your program must call the ProgramColorMapper operation either directly or indirectly. The operation is implemented in a hardware-independent manner and has run time checks for the support of each workstation type.

All procedures in the standard operating system library that manipulate color call ProgramColorMapper. The SetAlphaColorDefault procedure, for example, calls ProgramColorMapper to establish a default set of colors. SetAlphaColorDefault is called by the Executive as well as by routines in the graphics library. If you don't need to customize colors you will use, your program can use the default values. Three other color manipulating procedures are documented in the *CTOS Procedural Interface Reference Manual* for backwards compatibility only. These are

SetStyleRam
LoadColorStyleRam
SetStyleRamEntry

Because ProgramColorMapper performs all the functions provided by these procedures, it is recommended that you use ProgramColorMapper in all new programs.

Note that when an application returns control to the Executive, color palette values that were set within the application are reset to the current Executive default color palette values. (Other screen attributes, such as font type, resolution, number of lines and columns, reverse video, and character attributes, are also reset to the Executive default values when control returns to the Executive.)

ProgramColorMapper

You use ProgramColorMapper to perform the following functions:

- To enable or disable an alphanumeric or graphics display.
- To display alphanumeric characters in customized colors along with the standard alphanumeric character attributes (half bright, underline, reverse video, blinking, bold, and struck-through). Once you have defined the colors you want to use by calling ProgramColorMapper, you can assign them to the alphanumeric characters you intend to display using VAM or VDM operations such as PutFrameAttrs, PutFrameChars, and PutFrameCharsAndAttrs.
- To define color for use in graphics displays. If you don't want to use the default color palettes set up by routines in the graphics library, you can use ProgramColorMapper to define your own palettes for use in subsequent calls to graphics library procedures that manipulate the graphics bit map.

Once you have defined colors with ProgramColorMapper, you can redefine them by selecting different palettes or different palette entries or by changing the definition of existing entries.

ProgramColorMapper Structures

Through the parameters you provide to ProgramColorMapper, you set up the two types of color structures: a control structure and one or more palettes.

The *control structure* allows you to specify whether you want to enable graphics or alphanumeric mode and whether you want to enable color for either mode. In addition, with the ProgramColorMapper operation

available in Standard Software, Version 12.0 and higher, you can specify either of two palette formats. The *single-palette format* sets up a single palette for all color uses; the *three-palette format* allows you use of one to three palettes. For all hardware types, either format can be used, even if the actual color palette hardware is programmed differently.

Each *palette* contains entries that define color for alphanumeric, graphics, and/or background color. Each palette entry defines a color and its intensity. In "Application Notes," later in this section, you will see how color values defined in the palettes combine with the bits in the attribute byte to define the color and attributes of alphanumeric characters.

ProgramColorMapper

- maintains the control structure and palettes
- programs the hardware for the palette(s)
- programs the palette(s) according to the specifications you establish in the control structure

The single-palette format is described first because it provides greater flexibility. Because there is very little difference in the fields of the control structure for either palette format, both control structures are described in "Control Structure," later in this section.

Single-Palette Format

Using the single-palette format, the color palette is a single array of 3-word entries.

The words in each entry define color as follows:

Bits	**47–32**	**31–16**	**15–0**
	Red	Green	Blue

The palette can contain as many entries as the hardware will support (currently up to 256). Through the control structure for this palette format, you can define certain groups of entries for alphanumeric or background color. (See "Control Structure," for details.) Furthermore, any of the entries can be used for graphics color.

To define the intensity of a color, you specify a value in the range of 0 to 1000. Each unit increment in this range represents a 1/10% increase in intensity up to 100% intensity. For example, to display a color at 50% of its intensity, you can specify a value of 500. A value of 0 means no intensity.

Figure 3-2 shows six entries in an example palette.

(Red)	(Green)	(Blue)	
0	1000	0	0
0	500	0	1
0	0	1000	2
0	0	500	3
1000	0	0	4
. . .			. . .
500	0	0	n

2392.3–2

Figure 3-2. Example Palette in the Single-Palette Format

In the figure from top to bottom, the palette entries define the following colors:

full bright green
half-bright green
full bright blue
half-bright blue
full bright red
half-bright red

Each palette entry is identified by an index value. (See the numbers to the right of the entries in the figure.) Index 0 corresponds to the first entry, 1 to the second, and so forth. In a call to ProgramColorMapper, you specify the index value of the first palette entry you want to define in the control structure field *wIndexStart*. (See "Control Structure," later in this section.)

Advantages Over Three-Palette Format

There are several advantages to using the single-palette format over the three-palette. A few of these are noted here.

Because intensities can be specified in a broad spectrum, this format allows you to be very specific about the intensity of a color. If the hardware supports the intensity, the color intensity is displayed the way you specified. Otherwise the hardware displays the supported intensity closest to the one you chose. If, for example, the hardware supported three levels of intensity, you could specify 330, 670, and 100 to achieve 1/3, 2/3, and full intensity color, respectively. If you specified 256, the resulting color would be displayed at 1/3 intensity.

Furthermore, it is relatively easy to specify intensity values: you simply specify percentages. As you will see, this is more user friendly than specifying intensities using the three-palette format. The three-palette format requires that you manipulate bits.

In addition to the wide range of intensities offered by this format, the control structure allows you to use palette entries interchangeably for different applications.

Three-Palette Format

With this format, color can be defined in three palettes: one palette is an array of alphanumeric character color (*rgbAlpha*) and the other two (*rgbGraphics1* and *rgbGraphics2*) are arrays of graphics colors. Each palette in this format consists of 8 one-byte entries, as shown in the following structure:

Offset	Field	Size (Bytes)
0	rgbAlpha	8
8	rgbGraphics1	8
16	rgbGraphics2	8

Using the three-palette format in a call to ProgramColorMapper, you can assign colors to any of the following:

- a single alphanumeric palette (the first 8 bytes of the structure)
- the alphanumeric and the first graphics palette, rgbGraphics1 (the first 16 bytes of the structure)
- all three palettes (all 24 bytes)

The use of these palettes is described below.

Each one-byte palette entry is defined as follows:

Bit	7	6	5	4	3	2	1	0
	A1	A0	R1	R0	G1	G0	B1	B0

where

Letter	Description
A	Undefined
R	Red
G	Green
B	Blue

You specify the intensity of a color by manipulating the bits for that color (such as green, G1 G0)

where

Bit Values	Description
11	Full Intensity
10	2/3 Intensity (normally used for half bright)
01	1/3 Intensity
00	Off

Color Structure

For each of the palette formats, there is a corresponding control structure. The structure allows programs to set the color in any of the palettes and to turn the alphanumeric character map and the graphics bit map on or off independently.

The control structure for the three-palette format is shown below.

Offset	Field	Size (Bytes)
0	GraphicsPalette	1
1	AlphaEnabled	1
2	AlphaColorEnabled	1
3	GraphicsEnabled	1
4	GraphicsColorEnabled	1

Using the three-palette control structure, programs may switch the graphics bit map to use either of the two graphics palettes. This is done by specifying a value for the field *GraphicsPalette*. (See "Field Descriptions" for details.)

The format of a single-palette control structure is an extension of the structure shown above. It contains the following additional fields:

Offset	Field	Size (Bytes)
5	bFormat	1
6	wIndexStart	2

Field Descriptions

If ProgramColorMapper is called with a 0 value in any of the control structure fields, that field is left unchanged. The meanings of the control structure fields are described below:

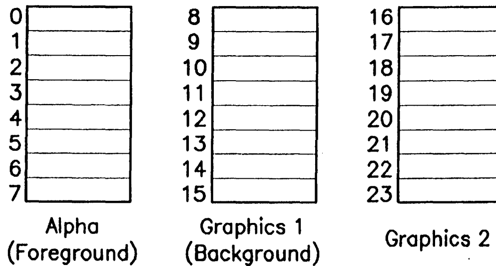
GraphicsPalette — Has no meaning with the single-palette format. With the three-palette format, a value of 1 selects the first graphics palette (*rgbGraphics1*), and a value of 2 selects the second (*rgbGraphics2*).

In the next four fields, a value of 1 means enabled, and a value of 2 means disabled.

AlphaEnabled — A value of 1 turns on the alphanumeric character map.

AlphaColorEnabled — With the three-palette format, a value of 1 means the alpha palette can be used for color selection. With the single-palette format, a value of 1 means that palette index entries 8 through 15 can be used for alphanumeric color.

On workstations with background color support, you can set the flag *fBackgroundColor* to TRUE in a call to the ResetVideoGraphics operation to use a background palette. With the three-palette format, the first graphics palette (index entries 8 through 15) is used for background color (see Figure 3-3). With the single-palette format, index entries 0 through 7 are used for background color. (See the example provided later in this section for a detailed description showing how the index entries are used as background colors). Specifying FALSE for *fBackgroundColor* allows use of only entry 0 (single-palette format) or entry 8 (three-palette format) for background color.



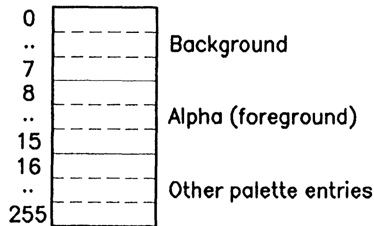


Figure 3-3. Three-Palette Format and Single-Palette Format

A value of 2 causes the default screen colors to be displayed (for example, green foreground with black background), and all normal character attributes are enabled.

NOTE: When AlphaColorEnabled *is set to 1, the normal character attribute for half bright has its meaning changed. (For details, see "Application Notes," later in this section.)*

GraphicsEnabled A value of 1 turns on the graphics bit map.

GraphicsColorEnabled With the three-palette format, a value of 1 means either graphics palette can be used for color selection. With the single-palette format, any of the palette entries can be used for graphics color.

bFormat

A value of 1 turns on the single-palette format. A value of 0 (or a control structure size of less than 6 bytes) causes ProgramColorMapper to return the three-palette format, even if the palette was previously defined in the single-palette format. In such a case, any value specified for *wIndexStart* (described below) is ignored.

wIndexStart

Specifies the index value of the first palette entry to be defined or read back using ProgramColorMapper.

If *AlphaEnabled* equals 1 and *AlphaColorEnabled* equals 1, the *fBackgroundColor* flag in the ResetVideoGraphics operation affects how background colors are chosen, as shown in "Example" below.

When *AlphaEnabled* equals 1 and *AlphaColorEnabled* equals 2, then the system default values are used for foreground, background, and half-bright, and the color palette is ignored. The same condition results when *GraphicsEnabled* equals 1 and *ColorGraphicsEnabled* equals 2.

Enabling Background Color Capability

Use the QueryVideo operation to determine the current setting of the *fBackgroundColor* flag.

Use the ResetVideoGraphics operation to enable and disable background color. Several other factors also determine whether background color is enabled or disabled.

1. Workstation hardware type. Some workstation configurations support background color and some do not. If your workstation configuration does not support background color, the settings described here have no effect.

2. If your workstation hardware does support background color (for example, with VGA-equipped workstations), you also need to enable background color mode explicitly by adding the following entry to your Config.sys file:

   :EVBackgroundOff: No

The default (Yes) disables background color mode.

Once you have enabled background color mode with the above *Config.sys* entry (and if your hardware supports it), you can turn the mode on and off with calls to ResetVideoGraphics. If selected applications need to use background color, it is recommended that you enable background color mode at the system level (with *Config.sys*) and then allow the individual application to turn background color on and off with ResetVideoGraphics.

Single-Palette Example

The following example illustrates how *fBackgroundColor* affects the background colors. Entries 8 through 15 make up the alphanumeric palette (foreground), and entries 0 through 7 make up Graphics Palette 1 (background). Assume colors are assigned to each entry as shown in Figure 3-4.

Graphics Palette 1 (Background)

Entry	Color
0	green
1	half–green
2	red
3	half–red
4	blue
5	half–blue
6	yellow
7	half–yellow

Alphanumeric Palette (Foreground)

Entry	Color
8	red
9	green
10	blue
11	yellow
12	black
13	magenta
14	cyan
15	white

512.3–4

Figure 3-4. Sample Single-Palette Format

Using the single-palette format, with *fBackgroundColor* set to TRUE, you obtain the following color combinations for the above palette. The notation used here is *foreground_color/background_color.*

red foreground	/	green background
green foreground	/	half-green background
blue foreground	/	red background
yellow foreground	/	half-red background
black foreground	/	blue background
magenta foreground	/	half-blue background
cyan foreground	/	yellow background
white foreground	/	half-yellow background

With *fBackgroundColor* set to FALSE, using the above palette, you obtain each foreground value (entries 8 through 15) with entry 0 (green in this case) as the background color in all cases.

Three-Palette Example

With the three-palette format, entries 0 through 7 make up the alphanumeric palette (foreground), and entries 8 through 15 make up Graphics Palette 1 (background). Assume colors are assigned to each entry as shown in Figure 3-5. Also assume that Graphics Palette 1 is selected by specifying 1 for the *GraphicsPalette* field of the control structure.

Entry	Alphanumeric Palette (Foreground)
0	green
1	half-green
2	red
3	half-red
4	blue
5	half-blue
6	yellow
7	half-yellow

Entry	Graphics 1 Palette (Background)
8	red
9	green
10	blue
11	yellow
12	black
13	magenta
14	cyan
15	white

512.3-5

Figure 3-5. Sample Three-Palette Format

Using the three-palette format, with *fBackgroundColor* set to TRUE, you obtain the following color combinations for the above palette. The notation used here is *foreground_color/background_color.*

green foreground	/	red background
half-green foreground	/	green background
red foreground	/	blue background
half-red foreground	/	yellow background
blue foreground	/	black background
half-blue foreground	/	magenta background
yellow foreground	/	cyan background
half-yellow foreground	/	white background

With *fBackgroundColor* set to FALSE, using the above palette, you obtain each foreground value (entries 0 through 7) with entry 8 as the background color in all cases.

Workstations with a 16-Entry Palette

Even if your workstation only supports a 16-entry palette, you can use the single palette format to define alphanumeric and graphics color by turning off the flag *fBackgroundColor* in a call to ResetVideoGraphics. Then with alpha color enabled, you can use index value 0 for background, index values 1 through 7 for graphics, and index values 8 through 15 for alphanumeric color.

Similarly, for the three-palette format, index values 0 through 7 would be used for alphanumeric color, index value 8 would be used as background, and index values 9 through 15 would be used for graphics color.

More Sample Palettes

The following examples show the use of the same palette with different settings for the control structure fields. These examples are for the three-palette format. Figure 3-6 shows the palette assignments for Case 1.

Alpha (Foreground)		Graphics 1 (Background)		Graphics 2 (Background)	
0	white	8	black	16	gray
1	red	9	brown	17	blue
2	green	10	half-cyan	18	yellow
3	blue	11	half-red	19	half-yellow
4	cyan	12	white	20	half-brown
5	orange	13	blue	21	red
6	magenta	14	red	22	white
7	yellow	15	green	23	black

512.3-6

Figure 3-6. Case 1 Palette Settings

Case 1: Condition A

Case 1 focuses on how changes to the *AlphaColorEnabled* and *GraphicsPalette* fields affect the resulting background colors used. Assume that the control structure fields are set as follows (where 1 means enabled, 2 means disabled, 0 means does not change, and x means don't care):

GraphicsPalette	*1*
AlphaEnabled	*1*
AlphaColorEnabled	*1*
GraphicsEnabled	*2*
GraphicsColorEnabled	*x*

If *fBackgroundColor* is set to FALSE in this case, then Alpha is used as the foreground palette, and Graphics 1 is the background palette (Graphics 2 is not used). Since background color is off, only entry 0 of Graphics 1 is used for background color. This value is white for white background monitors, and black for black background monitors. In this case, we see white characters over black background. The complete set of resulting colors for Condition A is shown below.

white/black
red/black
green/black
blue/black
cyan/black
orange/black
magenta/black
yellow/black

Case 1: Condition B

Assume *fBackgroundColor* is set to TRUE and the control structure fields are set the same as described in Condition A. In this case, Graphics 1 is used for background colors, as shown below.

white/black
red/brown
green/ 1/2 cyan
blue/ 1/2 red
cyan/white
orange/blue
magenta/red
yellow/green

Case 1: Condition C

If *fBackgroundColor* is TRUE and we change the value of the *GraphicsPalette* to 2, Graphics 2 is used for background colors, as shown below.

white/gray
red/blue
green/yellow
blue/ 1/2 yellow
cyan/ 1/2 brown
orange/red
magenta/white
yellow/black

Case 1: Condition D

Finally, assume that the control structure fields are set as follows (*AlphaColorEnabled* is OFF):

GraphicsPalette	*1*
AlphaEnabled	*1*
AlphaColorEnabled	*2*
GraphicsEnabled	*2*
GraphicsColorEnabled	*x*

If *fBackgroundColor* is FALSE as well, then the background color is one preset color chosen by the system. (If *fBackgroundColor* were TRUE, there would be eight preset colors chosen by the system.)

white/preset system color
red/preset system color
green/preset system color
blue/preset system color
cyan/preset system color
orange/preset system color
magenta/preset system color
yellow/preset system color

Case 2: Condition A

Case 2 focuses on how changes to the *GraphicsColorEnabled* field affect the resulting background colors used.

Assume that the control structure fields are set as follows (where 1 means enabled, 2 means disabled, 0 means does not change, and x means don't care):

GraphicsPalette	*2*
AlphaEnabled	*2*
AlphaColorEnabled	*x*
GraphicsEnabled	*1*
GraphicsColorEnabled	*2*

	Bitmap Memory (Foreground)
0	orange
1	green
2	yellow
3	blue
4	cyan
5	magenta
6	brown
7	black
	. . .

	Graphics 1 (Background)
8	black
9	brown
10	half-cyan
11	half-red
12	white
13	blue
14	red
15	green

	Graphics 2 (Background)
16	gray
17	blue
18	yellow
19	half-yellow
20	half-brown
21	red
22	white
23	black

512.3-7

Figure 3-7. Case 2 Palette Settings

If *fBackgroundColor* is set to FALSE in this case (where *GraphicsColorEnabled* is also set to FALSE), then the background values are preset colors chosen by the system, as follows:

orange/preset system color
green/preset system color
yellow/preset system color
blue/preset system color
cyan/preset system color
magenta/preset system color
brown/preset system color
black/preset system color

Case 2: Condition B

Next, assume that *fBackgroundColor* is set to FALSE and the control structure fields are set as follows (*GraphicsColorEnabled* is now TRUE):

GraphicsPalette	*2*
AlphaEnabled	*2*
AlphaColorEnabled	*x*
GraphicsEnabled	*1*
GraphicsColorEnabled	*1*

In this case, the resulting background color is the first entry of the specified graphics palette (here, Graphics 2).

orange/gray
green/gray
yellow/gray
blue/gray
cyan/gray
magenta/gray
brown/gray
black/gray

Case 2: Condition C

Finally, assume that *fBackgroundColor* is set to TRUE and the control structure fields are set as follows (*GraphicsColorEnabled* is TRUE):

GraphicsPalette	*2*
AlphaEnabled	*2*
AlphaColorEnabled	*x*
GraphicsEnabled	*1*
GraphicsColorEnabled	*1*

In this case, the resulting background colors are chosen in a similar manner to those in Case 1, except that bitmap characters are chosen from bitmap memory instead of from the Alpha palette.

orange/gray
green/blue
yellow/yellow
blue/ 1/2 yellow
cyan/ 1/2 brown
magenta/red
brown/white
black/black

Gray-Scale Monitors

If your monitor is not a color monitor, it is a "gray-scale" monitor. Gray-scale monitors interpret the entries in the color palette as shades of gray. Some monitors have very few gray scales (for example, some gray-scale monitors have only black and white; others have only green, half-green, and black). Other gray-scale monitors provide up to 64 shades of gray.

Use the QueryVideo operation to determine the specific capabilities of your monitor and graphics controller, as well as the number of intensities (gray scales) available on your monitor.

Both sample programs at the end of this chapter can be successfully run on gray-scale monitors that have a reasonable number of gray scales (16 or more) if you comment out the following lines:

```
if (!pVidInfo.fColorMonitor)
   ErrorExitString(0, "This workstation does not support
        color.", 40);
```

Systems with gray-scale monitors work in the same way as do color systems, except that gray shades are calculated based on red, green, and blue values. For systems with fewer than three gray scales, the fields *AlphaColorEnabled* and *GraphicsColorEnabled* in the Color Control structure are always 2 (disabled). For systems with fewer than four gray scales, the field *AlphaColorEnabled* in the Color Control structure is always 2 (disabled).

Application Notes

The following paragraphs provide detailed information on how to combine alphanumeric color with the values in the attribute byte, how to define bitmap color, and how to set color priorities.

Combining Alphanumeric Color With the Values in the Attribute Byte

On monochrome workstations or color workstations with color disabled, the attribute assigned to a character with PutFrameAttrs operation is defined as follows:

Bit	Attribute
0	Half bright
1	Underline
2	Reverse video
3	Blinking
4	Bold
5	Struck-through
6-7	not used

On a workstation with alphanumeric color enabled, the attributes are redefined as follows:

Bit	Attribute
0	p0
1	Underline
2	Reverse video
3	Blinking
4	Bold
5	Struck-through
6	p1
7	p2

p2, p1, and p0 combine to select the index number of one of the palette entries for alphanumeric color (as shown by example in the next few paragraphs). Although the half-bright attribute defined by bit 0 is lost, it is possible to regain half bright by defining half-bright colors.

With either the single-palette or the three-palette format, a maximum of eight entries can define alphanumeric color. Therefore, if you intend to preserve the six character attributes, four alpha colors are possible. Half bright intensity can be defined using four of the entries and full color intensity, using the remaining entries.

Following is an example showing the full bright and half-bright intensities of four colors using the three-palette format.

Palette Entry	Bit Assignments
0	00001100b (0Ch) (green)
1	00001000b (08h)
2	00110000b (30h) (red)
3	00100000b (20h)
4	00000011b (03h) (blue)
5	00000010b (02h)
6	00111100b (3Ch) (yellow)
7	00101000b (28h)

To make a red character with this palette, you specify an attribute byte to PutFrameAttrs of 01000000b or 40h. The following shows how this is done.

Recall that bits 0, 6, and 7 of the attribute byte combine to select the color index number. Looking at the bits in the attribute byte from left to right, the color index is formed as shown:

Bits in attribute byte	7	6	5	4	3	2	1	0
Bits forming color index	^	^						^
Bit values for color index 2	0	1						0

Combining bits 7, 6, and 0, the resulting binary value 010b is the value of the color index. This value selects rgbAlpha(2) of the palette above. The value of rgbAlpha(2) is 00110000b, in which R1R0 is 11 or full bright.

Because no attribute bits are set (bits 1 through 5 for underline, reverse video, blinking, bold, and struck-through, respectively), the value of the attribute byte in binary is

```
0 1 0 0 0 0 0 0
```

In hexadecimal, this is 40h.

To make a red, blinking, reverse-video character with the above palette, you specify an attribute byte with bits 2 and 3 turned on as shown below:

Bits in attribute byte	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Bits forming color index	^	^						^
	0	1	0	0	0	0	0	0
Bit turned on for reverse video	0	0	0	0	0	1	0	0
Bit turned on for blinking	0	0	0	0	1	0	0	0
Resulting binary value	0	1	0	0	1	1	0	0

In this case the value of the attribute byte is 01001100b or 4Ch.

NOTE: With the control structure field AlphaColorEnabled *set to 1, the single-palette format uses entries in the range of 8 through 15 for alphanumeric color. (See "Field Descriptions," earlier in this section.)*

The call to PutFrameAttrs would be the same whether you defined the eight color entries using the three-palette or the single-palette format. Using the single-palette with *AlphaColorEnabled* set to 1, the combined value of the color bits (2) would select entry 10, the third alphanumeric palette entry.

Defining Bitmap Color

For bitmap color, the graphics planes are combined for each pixel and are used as an index to the correct palette entry for the pixel color.

With the three-palette structure, the resulting value is the index of the entry in the graphics palette (either *rgbGraphics1* or *rgbGraphics2*, depending on the value of *GraphicsPalette* in the color control structure). With the single-palette format, *GraphicsPalette* has no meaning. The value directly indexes the entry defining the color.

NOTE: ResetVideo leaves the three palettes and the value of GraphicsPalette *untouched and resets* AlphaEnabled *and* GraphicsEnabled *to 0. SetScreenVidAttrs sets or resets* AlphaEnabled *and leaves the palettes,* GraphicsPalette, *and* GraphicsEnabled *untouched.*

Color Priorities

As indicated earlier in this section, the single-palette format allows you to use any of the palette entries for graphics color. With hardware that allows you to affect pixels using the alphanumeric character map or the graphics bit map, there can be some confusion over which color would actually be displayed. A priority must be set to determine the color displayed. This is the subject of the following example.

Assume that color is set up in the following way:

- The control structure field *AlphaColorEnabled* is the value 1.
- The flag *fBackgroundColor* was set to TRUE in a previous call to ResetVideoGraphics (entries 0 through 7 are background colors).
- The values of bits 0, 6, and 7 of the attribute byte combine to select palette entry 1.

P2	*P1*	*P0*
0	0	1

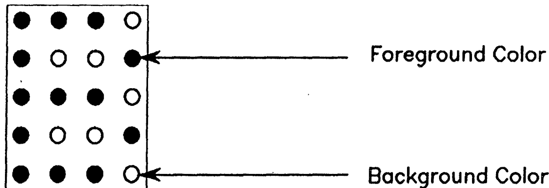
- The palette shown in Figure 3-8 is defined using the single-palette format.

	Graphics Palette 1 (Background)		Alphanumeric Palette (Foreground)
0	black	8	red
1	green	9	blue
2	..	10	..
3	..	11	..
4	..	12	..
5	..	13	..
6	..	14	..
7	..	15	..

512.3–8

Figure 3-8. Sample Palette

Using this setup, the letter B (shown in Figure 3-9) is displayed to the video device by a call to PutFrameChars. Entry 1 (P2 P1 P0 = 001) indicates blue foreground and green background, since the alphanumeric palette is always foreground and the graphics palette (either 1 or 2) is background. The letter B is thus displayed with a blue foreground, and background pixels are in green.



2392.3–1

Figure 3-9. Color Priorities

Now assume that in addition to the above conditions, *GraphicsEnabled* and *GraphicsColorEnabled* are set to 1 and that you manipulate the pixels through the graphics bit map. If you attempt to display each pixel in the right-most column shown in the figure using palette entry 8 (red), all the background color pixels in this column would display as red. All the foreground pixels, however, would display as blue.

When combining alphanumerics and graphics to manipulate pixels, the priority (from highest to lowest) of the color displayed is

1. alphanumeric foreground
2. graphics
3. background

Programming Tips

The following paragraphs provide useful programming tips.

Avoid Use of Reverse Video and Graphics

Setting the screen to reverse video mode (with SetScreenVidAttrs) can cause confusing results when used in conjunction with graphics. When alphanumerics is cleared in normal video (not reverse video), nothing exists in the character map, and so it is transparent. This means that any graphics below will be visible.

When alphanumerics is cleared to reverse video, all of the character cells are filled up. Since alphanumerics overlays graphics, the graphics will not be visible. It is advised that SetScreenVidAttrs be used to ensure that reverse video is not used in conjunction with graphics.

Obtaining Single-Palette Format

When you use ProgramColorMapper to return existing palette settings, the three-palette format is returned by default. To return the single-palette format, the *bFormat* control byte must be set to 1, and 8 bytes of new control information must be specified in the call to ProgramColorMapper. The number of control bytes must be 8. The *bFormat* control byte is not a permanent setting. It simply determines the treatment of the palette data that is passed or returned.

Avoid Combining Alpha Background with Graphics

Since palette entries are shared between the alpha background color palette and the graphics palette, the use of alpha background and graphics together is not advised. When background color mode is set and both alpha and graphics are used, graphics palette entries will be treated as alpha background colors and may produce unwanted background colors.

Program Examples

The following program examples show how to use the ProgramColorMapper operation to display color using the three-palette and single-palette formats. Both sample programs can be successfully run on gray-scale monitors that have a reasonable number of gray scales (16 or more) if you comment out the following lines:

```
if (!pVidInfo.fColorMonitor)
   ErrorExitString(0, "This workstation does not support
                     color.", 40);
```

Three-Palette Format Example

```
/*   Program name:  ThreePalette.c

     Description:  This is a sample program that shows how three
     palette style color works for character modes.  In this example,
     only the alpha palette is used.
*/
#define Syslit
#include <CTOSTypes.h>

#define CheckErc
#define ErrorExit
#define ErrorExitString
#define ProgramColorMapper
#define PutFrameAttrs
#define PutFrameChars
#define QueryVideo
#define ReadKbdDirect
#define ResetFrame
#define SetAlphaColorDefault
#include <CTOSLib.h>

/*   Color literals;  NOTE that these are constructed using the
     'three palette'format:
          Al A0 Rl R0 Gl G0 Bl B0
     For example, the Magenta color is formed by:
          Al Al Rl R0 Gl G0 Bl B0
           0  0  1  1  0  0  1  1 = 33 Hex
*/
```

Listing 3-1. ThreePalette.c. (Page 1 of 7)

```
#define lBlue         0x3
#define lHalfBlue     0x2
#define lMagenta      0x33
#define lHalfMagenta  0x22
#define lAmber        0x38
#define lOrange       0x34
#define lSalmon       0x25
#define lGrey         0x15
#define lReverseVideo 4
#define lBlinking     8

typedef struct
{
    Byte              Level;
    Byte              nLinesMax;
    Byte              nColsNarrow;
    Byte              nColsWide;
    Byte              GraphicsVersion;
    Word              nPixelsHigh;
    Word              nPixelsWide;
    Word              saGraphicsBoard;
    Word              ioPort;
    Word              wBytesPerLine;
    Byte              nCharHeight;
    Byte              nCharWidthNarrow;
    Byte              nCharWidthWide;
    Pointer           pBitmap;
    Pointer           pFont;
    Byte              bModuleType;
    Byte              bModulePos;
    Word              wModuleEar;
    Word              nYCenter;
    Word              nXCenterNarrow;
    Word              nXCenterWide;
    Word              wxAspect;
    Word              wyAspect;
    Byte              bPlanes;
    Byte              fColorMonitor;
    Word              nColors;
    Byte              fBackgroundColor;
    Byte              fHardwareCharMap;
    Word              nAlternateLinesMax;
    Word              nAlternateCharHeight;
    Word              wVidRelease;
    Word              wVidVersion;
    Byte              Reserved[45];
} VidInfoType;
```

Listing 3-1. ThreePalette.c. (Page 2 of 7)

```
typedef struct
{
    Byte                bGraphicsPalette;
    Byte                bAlphaEnabled;
    Byte                bAlphaColorEnabled;
    Byte                bGraphicsEnabled;
    Byte                bGraphicsColorEnabled;
    Byte                bFormat;
    Word                wIndexStart;
} ClrCtrlType;

void main ()
{
    Byte                rgbNewPalette[8],
                        rgbOldPalette[8],
/*
    Color Attributes:

    As an example of how to interpret the color attribute byte, let's
    look at rgbColorAttrs[6]:

    If you want to make some text yellow (using default colors), you
    need to set the color index in the attribute byte to six (6).  The
    bits in the attribute byte that form the color index are bits 0,
    6, and 7.  Bits 1 through 5 are used for the other character
    attributes (underline, reverse video, blinking, bold, and struck-
    through):

    Attribute byte:         76543210
    Color index formed      ^^     ^
    from these bits:        ||     |
    To get index 6          11xxxxx0 (bit 0 is the 1's place, bit 6
    is the 2's place and bit 7 is the 4's place)
    To get the attr mask 11000000 = 0xC0 (hex).  Selects color 6 and
    no other attributes.

    When using color you lose the half-bright attribute.  If you have
    half-bright colors defined, then you can use those has the 'half-
    bright attribute'.
*/
```

Listing 3-1. ThreePalette.c. (Page 3 of 7)

```
rgbColorAttrs[8] =
{
     0,
     1,
     0x40,
     0x41,
     0x80,
     0x81,
     0xC0,
     0xC1
}, bChar;
char *rgbTextDef[] =
{
     "This is green            ",
     "This is half-bright green",
     "This is cyan             ",
     "This is half-bright cyan ",
     "This is white            ",
     "This is half-bright white",
     "This is yellow           ",
     "This is red              "
},
*rgbTextNew[] =
{
     "This is blue               ",
     "This is half-bright blue   ",
     "This is Magenta            ",
     "This is half-bright Magenta",
     "This is amber              ",
     "This is orange             ",
     "This is salmon             ",
     "This is grey               "
};
Word
     iCol,
     iLine;
ClrCtrlType
     NewControl,
     OldControl;
VidInfoType
     pVidInfo;


CheckErc (QueryVideo (&pVidInfo, 100));
```

Listing 3-1. ThreePalette.c. (Page 4 of 7)

```
/*
    First check if we can use color.
    NOTE:  This check is not quite correct considering that there
    are analog monochrome monitors that can support shades of
    grey.  Since this is an example of color, we won't try to run on
    monochrome monitors.
*/

    if (!pVidInfo.fColorMonitor)
        ErrorExitString (0, "This workstation does not support
        color.", 40);

/*
    Save old color palette and control so that we can reset upon exit.
    We assume that we are being 'run' from the Executive.
    The Executive uses the 'three palette' format so we only need
    to get 8 bytes for the Color Palette and 5 bytes for the
    Control Structure.
*/

    CheckErc (ProgramColorMapper (rgbNewPalette, 0, &NewControl,
    0,rgbOldPalette, 8, &OldControl, 5));

/*
    Set up default alpha color palette.
*/

    CheckErc (SetAlphaColorDefault (0));

/*
    Display some color text using default palette.
*/
    CheckErc (ResetFrame (0));
    for (iLine = 0, iCol = 0; iLine < 8; iLine++, iCol++)
    {
        CheckErc (PutFrameChars (0, iCol, iLine,
        rgbTextDef[iLine], 25));
        CheckErc (PutFrameAttrs (0, iCol, iLine,
        rgbColorAttrs[iLine], 25));
    }
    CheckErc (ReadKbdDirect (0, &bChar));

/*
    Now use color and reverse video (default palette);  Note that we
    are ORing the color attribute byte with the Reverse Video
    attribute (4)
*/
```

Listing 3-1. ThreePalette.c. (Page 5 of 7)

```
    CheckErc (ResetFrame (0));
    for (iLine = 0, iCol = 0; iLine < 8; iLine++, iCol++)
    {
        CheckErc (PutFrameChars (0, iCol, iLine,
        rgbTextDef[iLine], 25));
        CheckErc (PutFrameAttrs (0, iCol, iLine,
        rgbColorAttrs[iLine] | lReverseVideo, 25));
    }
    CheckErc (ReadKbdDirect (0, &bChar));

/*
    Now let's change the palette
*/

    rgbNewPalette[0] = lBlue;
    rgbNewPalette[1] = lHalfBlue;
    rgbNewPalette[2] = lMagenta;
    rgbNewPalette[3] = lHalfMagenta;
    rgbNewPalette[4] = lAmber;
    rgbNewPalette[5] = lOrange;
    rgbNewPalette[6] = lSalmon;
    rgbNewPalette[7] = lGrey;
    NewControl.bGraphicsPalette = 2;
    NewControl.bAlphaEnabled = 1;
    NewControl.bAlphaColorEnabled = 1;
    NewControl.bGraphicsEnabled = 2;
    NewControl.bGraphicsColorEnabled = 2;
    CheckErc (ProgramColorMapper (rgbNewPalette, 8, &NewControl,
    5, rgbOldPalette, 0, &OldControl, 0));

/*
    Display some color text using new palette.
*/

    CheckErc (ResetFrame (0));
    for (iLine = 0, iCol = 0; iLine < 8; iLine++, iCol++)
    {
        CheckErc (PutFrameChars (0, iCol, iLine,
        rgbTextNew[iLine], 27));
        CheckErc (PutFrameAttrs (0, iCol, iLine,
        rgbColorAttrs[iLine], 27));
    }
    CheckErc (ReadKbdDirect (0, &bChar));
```

Listing 3-1. ThreePalette.c. (Page 6 of 7)

```
/*
    Now use color and blinking attribute; Note that we are ORing the
    color attribute byte with the blinking attribute (8)
*/

    CheckErc (ResetFrame (0));
    for (iLine = 0, iCol = 0; iLine < 8; iLine++, iCol++)
    {
        CheckErc (PutFrameChars (0, iCol, iLine,
        rgbTextNew[iLine], 27));
        CheckErc (PutFrameAttrs (0, iCol, iLine,
        rgbColorAttrs[iLine] | lBlinking, 27));
    }
    CheckErc (ReadKbdDirect (0, &bChar));

/*
    Reset palette and control structure to original values.
*/
    CheckErc (ProgramColorMapper (rgbOldPalette, 8, &OldControl, 5,
    rgbOldPalette, 0, &OldControl, 0));

    ErrorExit (0);
}
```

Listing 3-1. ThreePalette.c. (Page 7 of 7)

Single-Palette Format Example

```
/*
    Program name:  NewPalette.c

    Description:  This is a sample program that shows how the single
    palette style color works for character modes.

    Caveats:  This is meant to be a sample program and should not be
    considered to be a 'real' program.  It is just a representation of
    basic character color concepts.  Also, this program was developed
    for use with the Metaware™ High C compiler for CTOS.  It will
    most likely require changes for other C compilers.
*/

#include <stdio.h>

#define Syslit
#include <CTOSTypes.h>
#define CheckErc
#define ErrorExit
#define ErrorExitString
#define InitCharMap
#define InitVidFrame
#define ProgramColorMapper
#define PutFrameAttrs
#define PutFrameChars
#define QueryVideo
#define ReadKbdDirect
#define ResetFrame
#define SetAlphaColorDefault
#define SetScreenVidAttr
#include <CTOSLib.h>

pragma Calling_Convention (CTOS_CALLING_CONVENTIONS);
extern  ErcType  ResetVideoGraphics  (Byte  nCols,  Byte  nLines,  Byte
fAttr,Byte bSpace, Pointer psMapRet,Word nPixelsWide, Word nPixelsHigh,
Word bPlanes, FlagType fBackgroundColor);

#define lReverseVideo  4

typedef struct {
    Byte     Level;
    Byte     nLinesMax;
    Byte     nColsNarrow;
    Byte     nColsWide;
    Byte     GraphicsVersion;
```

Listing 3-2. NewPalette.c. (Page 1 of 11)

```
    Word      nPixelsHigh;
    Word      nPixelsWide;
    Word      saGraphicsBoard;
    Word      ioPort;
    Word      wBytesPerLine;
    Byte      nCharHeight;
    Byte      nCharWidthNarrow;
    Byte      nCharWidthWide;
    Pointer   pBitmap;
    Pointer   pFont;
    Byte      bModuleType;
    Byte      bModulePos;
    Word      wModuleEar;
    Word      nYCenter;
    Word      nXCenterNarrow;
    Word      nXCenterWide;
    Word      wxAspect;
    Word      wyAspect;
    Byte      bPlanes;
    Byte      fColorMonitor;
    Word      nColors;
    Byte      fBackgroundColor;
    Byte      fHardwareCharMap;
    Word      nAlternateLinesMax;
    Word      nAlternateCharHeight;
    Word      wVidRelease;
    Word      wVidVersion;
    Byte      Reserved[45];
}   VidInfoType;

typedef struct
{
    Byte      bGraphicsPalette;
    Byte      bAlphaEnabled;
    Byte      bAlphaColorEnabled;
    Byte      bGraphicsEnabled;
    Byte      bGraphicsColorEnabled;
    Byte      bFormat;
    Word      wIndexStart;
}   ClrCtrlType;

typedef struct
{
    Word      wRed;
    Word      wGreen;
    Word      wBlue;
} RGBColorType;
```

Listing 3-2. NewPalette.c. (Page 2 of 11)

```
void main ()
{
    Byte     rgbOldPalette[8],
/*
    Color attribute byte.  As an example of how to interpret the color
    attribute byte, let's look at rgbColorAttrs[6]:

    If you want to make some text yellow, you need to set the color
    index in the attribute byte to six (6).  The bits in the attribute
    byte that form the color index are bits 0, 6, and 7.  Bits 1
    through 5 are used for the other character attributes (underline,
    reverse video, blinking,bold, and struck-through):

    Attribute byte:            76543210
        Color index formed     ^^     ^
        from these bits:       ||     |
        To get index 6         11xxxxx0 (bit 0 is the 1's place, bit 6
        is the 2's place and bit 7 is the 4's place)
        To get the mask        11000000 = 0xC0 (hex)

        When using color you lose half-bright attribute.  If you have
        half-bright colors defined, then you can use those has the
        'half-bright attribute'.
*/

    rgbColorAttrs[8] =
    {
        0,
        1,
        0x40,
        0x41,
        0x80,
        0x81,
        0xC0,
        0xC1
    },
    bChar;

    FlagType
        fEV,
        fGC001,
        fBackgroundColor;
```

Listing 3-2. NewPalette.c. (Page 3 of 11)

```
char
*rgbText1[] =
{
     "This is green            ",
     "This is half-bright green",
     "This is cyan             ",
     "This is half-bright cyan ",
     "This is white            ",
     "This is half-bright white",
     "This is yellow           ",
     "This is red              "
},
*rgbText2[] =
{
     "Text green/Background black                            ",
     "Text half-bright green/Background grey                 ",
     "Text cyan/Background blue                              ",
     "Text half-bright cyan/Background half-bright blue      ",
     "Text white/Background magenta                          ",
     "Text half-bright white/Background half-bright magenta",
     "Text yellow/Background half-bright red                 ",
     "Text red/Background half-bright yellow                 "
},
*rgbText3[] =
{
     "Text black/Background green                            ",
     "Text grey/Background half-bright green                 ",
     "Text blue/Background cyan                              ",
     "Text half-bright blue/Background half-bright cyan      ",
     "Text magenta/Background white                          ",
     "Text half-bright magenta/Background half-bright white",
     "Text half-bright red/Background yellow                 ",
     "Text half-bright yellow/Background red                 "
};
Word     iCol,
         iLine,
         nCols,
         nLines,
         sCharMap;
ClrCtrlType
         NewControl,
         OldControl;
RGBColorType
         rgColorPalette[16];

VidInfoType
         pVidInfo;
```

Listing 3-2. NewPalette.c. (Page 4 of 11)

```
    CheckErc (QueryVideo (&pVidInfo, 100));
    nCols = pVidInfo.nColsNarrow;
    nLines = pVidInfo.nLinesMax;

    if ((pVidInfo.wVidRelease < 3) && (pVidInfo.wVidVersion < 1))
        ErrorExitString (0, "This program requires VAM version 3.1 or
        later.", 47);

/*
    First check if we can use color.
    NOTE:  This check is not quite correct considering that there are
    analog monochrome monitors that can support shades of grey.  Since
    this is an example of color, we won't try to run on monochrome
    monitors.
*/

    if (!pVidInfo.fColorMonitor)
        ErrorExitString (0, "This workstation does not support
        color.", 40);

/*
    If we made it past the previous check, then check if we have
    Extended Video:
    Check if fBackgroundColor is already TRUE, that we can set 34
    lines and that the resolution is set to 0.  This almost assures us
    that we are Extended Video and not GC-x04.
    NOTE:  This should not cause any problems, since fBackgroundColor
    will always be FALSE for GC-001 and GC-x04 always has resolution
    set to some non-zero state (e.g., 720x348 or 1024x768)
*/

    fEV = (pVidInfo.fBackgroundColor && (pVidInfo.nAlternateLinesMax
    == 34) && (pVidInfo.nPixelsWide == 0) && (pVidInfo.nPixelsHigh ==
    0));

/*
    Make sure we know if we are on GC-001 with color monitor;
*/

    fGC001 = (pVidInfo.GraphicsVersion == 3);

/*
    Save old color palette and control so that we can reset upon exit.
    We assume that we are being 'run' from the Executive.  The
    Executive uses the 'three palette' format so we only need to get 8
    bytes for the Color Palette and 5 bytes for the Control Structure.
*/
```

Listing 3-2. NewPalette.c. (Page 5 of 11)

```
    CheckErc (ProgramColorMapper (0, 0, 0, 0, rgbOldPalette, 8,
    &OldControl, 5));

/*
    Start setting up new color palette;  We are making this color
    palette look like default color palette as set up by the
    SetAlphaColorDefault call.  NOTE that foreground colors start at
    index 8.

    Background color 0 - black
*/

    rgColorPalette[0].wRed = 0;
    rgColorPalette[0].wGreen = 0;
    rgColorPalette[0].wBlue = 0;

/*
    Background color 1 - Dark grey
*/
    rgColorPalette[1].wRed = 333;
    rgColorPalette[1].wGreen = 333;
    rgColorPalette[1].wBlue = 333;

/*
    Background color 2 - Blue
*/

    rgColorPalette[2].wRed = 0;
    rgColorPalette[2].wGreen = 0;
    rgColorPalette[2].wBlue =  1000;

/*
    Background color 3 - Half-bright Blue
*/
    rgColorPalette[3].wRed = 0;
    rgColorPalette[3].wGreen = 0;
    rgColorPalette[3].wBlue = 500;

/*
    Background color 4 - Magenta
*/

    rgColorPalette[4].wRed = 1000;
    rgColorPalette[4].wGreen = 0;
    rgColorPalette[4].wBlue = 1000;
```

Listing 3-2. NewPalette.c. (Page 6 of 11)

```
/*
    Background color 5 - Half-bright Magenta
*/

    rgColorPalette[5].wRed = 500;
    rgColorPalette[5].wGreen = 0;
    rgColorPalette[5].wBlue = 500;

/*
    Background color 6 - Half-bright red
*/

    rgColorPalette[6].wRed = 500;
    rgColorPalette[6].wGreen = 0;
    rgColorPalette[6].wBlue = 0;

/*
    Background color 7 - Half-bright yellow
*/

    rgColorPalette[7].wRed = 500;
    rgColorPalette[7].wGreen = 500;
    rgColorPalette[7].wBlue = 0;

/*
    Foreground color 0 - Green
*/

    rgColorPalette[8].wRed = 0;
    rgColorPalette[8].wGreen = 1000;
    rgColorPalette[8].wBlue = 0;

/*
    Foreground color 1 - Half-bright green
*/

    rgColorPalette[9].wRed = 0;
    rgColorPalette[9].wGreen = 500;
    rgColorPalette[9].wBlue = 0;

/*
    Foreground color 2 - Cyan
*/

    rgColorPalette[10].wRed = 0;
    rgColorPalette[10].wGreen = 1000;
    rgColorPalette[10].wBlue =  1000;
```

Listing 3-2. NewPalette.c. (Page 7 of 11)

```
/*
     Foreground color 3 - Half-bright cyan
*/

     rgColorPalette[11].wRed = 0;
     rgColorPalette[11].wGreen = 500;
     rgColorPalette[11].wBlue = 500;

/*
     Foreground color 4 - White
*/

     rgColorPalette[12].wRed = 1000;
     rgColorPalette[12].wGreen = 1000;
     rgColorPalette[12].wBlue = 1000;

/*
     Foreground color 5 - Half-bright white
*/

     rgColorPalette[13].wRed = 500;
     rgColorPalette[13].wGreen = 500;
     rgColorPalette[13].wBlue = 500;

/*
     Foreground color 6 - Yellow
*/

     rgColorPalette[14].wRed = 1000;
     rgColorPalette[14].wGreen = 1000;
     rgColorPalette[14].wBlue = 0;

/*
     Foreground color 7 - Red
*/
     rgColorPalette[15].wRed = 1000;
     rgColorPalette[15].wGreen = 0;
     rgColorPalette[15].wBlue = 0;

/*
     Enable alpha color only and set up the color palette.
*/
```

Listing 3-2. NewPalette.c. (Page 8 of 11)

```
    NewControl.bGraphicsPalette = 2;
    NewControl.bAlphaEnabled = 1;
    NewControl.bAlphaColorEnabled = 1;
    NewControl.bGraphicsEnabled = 2;
    NewControl.bGraphicsColorEnabled = 2;
    NewControl.bFormat = 1;
    NewControl.wIndexStart = 0;

/*
    Set new color palette.  Note that size of the color palette is:
    3 Words (Red, Green, Blue) times 2 bytes per word times the number
    of entries in the palette (16) = 96.
*/
    CheckErc (ProgramColorMapper (rgColorPalette, 96, &NewControl, 8,
    0, 0, 0, 0));

/*
    Note that we won't do the following with enhanced video (EV) since
    this part of the code assumes that fBackgroundColor is FALSE
    and makes the messages appear incorrect.
*/
    if (!fEV)
    {
/*
    Display some color text
*/
        CheckErc (ResetFrame (0));
        for (iLine = 0, iCol = 0; iLine < 8; iLine++, iCol++)
        {
            CheckErc (PutFrameChars (0, iCol, iLine, rgbText1[iLine],
            25));
            CheckErc (PutFrameAttrs (0, iCol, iLine,
            rgbColorAttrs[iLine], 25));
        }
        CheckErc (ReadKbdDirect (0, &bChar));

/*
    Now use color and reverse video.
*/
        CheckErc (ResetFrame (0));
        for (iLine = 0, iCol = 0; iLine < 8; iLine++, iCol++)
        {
            CheckErc (PutFrameChars (0, iCol, iLine, rgbText1[iLine],
            25));
            CheckErc (PutFrameAttrs (0, iCol, iLine,
            rgbColorAttrs[iLine] | lReverseVideo, 25));
        }
```

Listing 3-2. NewPalette.c. (Page 9 of 11)

```
            CheckErc (ReadKbdDirect (0, &bChar));
/*
        Set fBackgroundColor to TRUE (if supported) and reinitialize the
        character map
*/
            fBackgroundColor = !fGC001;
            CheckErc (ResetVideoGraphics (nCols, nLines, TRUE, 0x20,
            &sCharMap, 0, 0, 0, fBackgroundColor));
            CheckErc (InitVidFrame (0, 0, 0, nCols, nLines, 0, 0, 0,
            FALSE, FALSE));
            CheckErc (InitCharMap (0, sCharMap));
            CheckErc (SetScreenVidAttr (1, TRUE));

/*
        Set new color palette.  Note that size of the color palette is:
        3 Words (Red, Green, Blue) times 2 bytes per word times the number
        of entries in the palette (16) = 96.
*/

            CheckErc (ProgramColorMapper (rgColorPalette, 96, &NewControl,
            8, 0, 0, 0, 0));
        }
/*
        end of if (!fEV)
*/

/*
        Background color has no meaning on GC-001 so rather than confuse
        the person running the program, we won't even do this.  Again
        messages don't reflect what is displayed on the screen.
*/
        if (!fGC001)
        {
/*
        Display some color text
*/
            CheckErc (ResetFrame (0));
            for (iLine = 0, iCol = 0; iLine < 8; iLine++, iCol++)
            {
                CheckErc (PutFrameChars (0, iCol, iLine, rgbText2[iLine],
                53));
                CheckErc (PutFrameAttrs (0, iCol, iLine,
                rgbColorAttrs[iLine], 53));
            }
            CheckErc (ReadKbdDirect (0, &bChar));
```

Listing 3-2. NewPalette.c. (Page 10 of 11)

```
/*
    Now use color and reverse video.
*/

        CheckErc (ResetFrame (0));
        for (iLine = 0, iCol = 0; iLine < 8; iLine++, iCol++)
        {
            CheckErc (PutFrameChars (0, iCol, iLine, rgbText3[iLine],
            53));
            CheckErc (PutFrameAttrs (0, iCol, iLine,
            rgbColorAttrs[iLine] | lReverseVideo, 53));
        }
        CheckErc (ReadKbdDirect (0, &bChar));
    }   /* end of if (!fGC001) */

/*
    Reset to original state;
*/

    CheckErc (ProgramColorMapper (rgbOldPalette, 8, &OldControl, 5, 0,
    0, 0, 0));

    CheckErc (ResetVideoGraphics (0, 0, TRUE, 0x20, &sCharMap, 0, 0,
    0, FALSE));
    ErrorExit (0);

}
```

Listing 3-2. NewPalette.c. (Page 11 of 11)

4
Writing Partition-Managing Programs

Certain kinds of applications need the ability to "spawn" other programs. For example, some applications allow users to run utility programs in the background, while the application continues to run in the foreground.

One way to accomplish this is for the application to create a child partition and run the utility program in that partition. Then, when the utility program has terminated, the application deallocates the child partition.

This chapter discusses the methods available to accomplish that.

A Review of Partition Management Operations

As discussed in the *CTOS Operating System Concepts Manual*, the operating system provides several operations which aid in partition management. The Context Manager also includes a library of operations which can be used when the Context Manager is installed.

In general, if your program will always run when Context Manager is installed, use the Context Manager operations. However, if your program needs to run whether or not Context Manager is installed you must use the operating system partition management operations directly.

This chapter addresses the latter case, when a program must run whether or not the Context Manager is present. See your Context Manager documentation for more information about the programmatic interface to the Context Manager.

Creating a Partition and Loading a Program into It

Creating partitions is not particularly complicated, as long as you perform four basic steps in the correct order. The steps in creating a partition are:

1. Allocate memory and create the partition.
2. Swap its context into memory.
3. Set up the partition's environment.
4. Load a task into the partition.

The following sections discuss each of these steps in more detail. For an example of a complete partition-creating procedure, see the listings at the end of this chapter.

Creating a Partition and Swapping It into Memory

Listing 4-1 shows a code fragment that creates a partition and swaps it into memory. In protected mode, any partition can create other partitions. In real mode operating systems, only the primary partition can create other partitions.

```
erc = CreateBigPartition("MyChild", 7,(DWord)
        MemSize*64 /*parags*/, Par6, &userNumChild);
if(erc != ercOK)
   return(erc);

RetCleanup(SwapInContext(userNumChild));
```

Listing 4-1. Creating a Partition

The fragment first creates a partition using CreateBigPartition. Using CreateBigPartition allows the program to create a partition of any size, while CreatePartition limits partition size to one megabyte. Note, however, that CreateBigPartition works only on protected-mode workstations.

The created partition is of type 6, which allows the partition to be resized dynamically by the operating system whenever a program in the partition terminates. Also, note that the MemSize parameter represents a number of K bytes. This number must be multiplied by 64 to yield the paragraph count needed by the CreatePartition calls.

After the program has created the partition, it swaps the new partition into memory. This causes the operating system to verify that enough physical memory is available, and to allocate that memory to the newly-created partition. A program must perform a SwapInContext before it attempts to load a task into a newly-created partition, or the results will be undefined.

RetCleanup is a macro which causes the program to deallocate the partition in case of error. See the complete program listing at the end of this chapter if you want to see how the macro is defined.

Partition-managing programs must be extremely careful about deallocating partitions they create. If a partition-managing program exits without deallocating a partition it created, that partition cannot be destroyed without rebooting the system.

Setting Up the Program's Environment

After the partition has been created, the program must set up the partition environment for the task it plans to load. The amount of setup required depends on the resources the task will use.

In general, the minimal setup consists of initializing the new partition's virtual screen and Application System Control Block. This ensures that the program can write to its virtual screen, and can retrieve basic information about itself.

The code fragment in Listing 4-2 shows the initialization of a newly-created partition. Note that "Alt" requests are used for most of these tasks.

Alt requests allow a program to send requests on behalf of another user number. For each request the operating system serves, there is also an Alt version. The only difference between a regular request and an Alt request is that the Alt requests take an additional parameter, the user number for whom the request should be sent.

```
/* copy Vlpb and insert its address in the child
   partition's Ascb */
RetCleanup(GetpStructure(lGetpVLPB, 0, &pVlpbMine));
RetCleanup(AltAllocMemoryLL(userNumChild,
       *(Word *)pVlpbMine, &pVlpbChild));
memcpy(pVlpbChild, pVlpbMine, *(Word *)pVlpbMine);
RetCleanup(GetpStructure(lGetpAscb, userNumChild,
       &pAscbChild));
pAscbChild->pVlpb = pVlpbChild;

/* copy user name, signon password, and sys.cmds spec
   in new Ascb */
RetCleanup(GetpStructure(lGetpAscb, 0, &pAscbMine));
memcpy(pAscbChild->sbUserName, pAscbMine->sbUserName,
       SCOPYAREA);

/* sneakily replace my pVlpb with childs so I can get
 * params.  Have to do this cuz RgParam ops are not
 * requests, so no Alt versions available */
pTemp = pAscbMine->pVlpb;
pAscbMine->pVlpb = pVlpbChild;
/* function to get any parameters for the child */
RetCleanup(GetParams(pVlpbChild));
/* restore my Ascb */
pAscbMine->pVlpb = pTemp;

/* set child's exit run file */
RetCleanup(AltSetExitRunFile(userNumChild,
       "[Sys]<Sys>Exec.run", 18, NULL, 0, 0x80));

/* copy current path to new partition */
RetCleanup(GetUCB(&ucb, sizeof(ucb)));
RetCleanup(AltSetPath(userNumChild, ucb.rgbVol,
       ucb.cbVol, ucb.rgbDir, ucb.cbDir, ucb.rgbPswd,
       ucb.cbPswd));
if (ucb.cbPrefix)
   RetCleanup(AltSetPrefix(userNumChild, ucb.rgbPrefix,
       ucb.cbPrefix));
```

Listing 4-2. Initializing a New Partition's Environment (Page 1 of 2)

```
if (ucb.cbNode)
   RetCleanup(AltSetNode(userNumChild, ucb.rgbNode,
         ucb.cbNode));

/* initialize exec screen - font doesn't need loading,
    nor double bar */
RetCleanup( AltQueryVidHdw(userNumChild, &vidHdw,
            sizeof(vidHdw)));
cLines = vidHdw.nLinesMax;
cCols  = vidHdw.nColsNarrow;
RetCleanup(AltResetVideo(userNumChild, cCols, cLines,
         TRUE, 0, &sMap));
RetCleanup(AltInitVidFrame(userNumChild, 1, 0, 0,
         cCols, 2, 0, 0, 0, 0, 0));
RetCleanup(AltInitVidFrame(userNumChild, 2, 0, 2,
         cCols, 1, 0, 0, 0, 0, 0));
RetCleanup(AltInitVidFrame(userNumChild, 0, 0, 3,
         cCols, cLines - 3, 0, 0, 0, 0, 0));
RetCleanup(AltInitCharMap(userNumChild, 0, sMap));
RetCleanup(AltSetScreenVidAttr(userNumChild,1, 0xFF));
```

Listing 4-2. Initializing a New Partition's Environment (Page 2 of 2)

Loading the Program into the New Partition

Once the partition has been created and the environment has been set up, the partition-managing program can load a task into the new partition. A task simply consists of a run file. Calling LoadInteractiveTask or LoadPrimaryTask causes the operating system to read the specified run file from disk into memory, then to create and schedule a process for the new task.

When LoadInteractiveTask or LoadPrimaryTask return to the caller, the newly loaded program is in memory and scheduled for execution. Listing 4-3 shows a call to LoadInteractiveTask.

A partition-managing program must ensure that the run file can be loaded successfully before it attempts to load the task. If the operating system attempts to load the task but fails, the workstation may crash. In particular, this condition can occur when insufficient memory is specified for the partition and the Context Manager is not installed.

```
RetCleanup(LoadInteractiveTask(userNumChild,
        pRunFileName, strlen(pRunFileName), NULL, 0,
        StdPriority, FALSE));
RetCleanup(SwapInContext(userNumChild));
```

Listing 4-3. Loading a Task for Execution

This condition occurs because of the way the operating system interacts with the Context Manager. If the operating system's attempt to load a task into a partition fails, it checks whether the NotifyCM request is served. If the request is served, the operating system sends a NotifyCM request asking the Context Manager to remove the partition. However, if the NotifyCM request is not served, the operating system calls Crash.

Finding the Program's Termination Status

As shown by the last three sections, creating a partition and executing a program in it is not particularly difficult. Removing a partition is equally simple, and is discussed later in this chapter. The difficult aspect of partition management is discovering when the program terminated and what its termination status was.

The Child-Termination Question

There is no operating system request that allows a program to determine the termination status of another program. A program can examine another program's ASCB to discover the most recent exit status code, but that provides only half the picture. How can it know whether the program in which it is interested has terminated? If that program is still active, the status code in the ASCB is irrelevant.

Also, the partition-managing program generally does not just want to discover at some future time whether its child has finished executing. It wants to be notified immediately when the program in its child partition terminates.

A Solution – Defining a Termination Request

Because there is no predefined operating system request for it, there is no simple way to determine when a child terminates. It can be done, however, by defining your own termination request.

Whenever any program terminates, the operating system issues termination requests for it. These termination requests notify other programs (usually system services) that the program is terminating. This notification allows system services to perform any needed cleanup functions if the terminating program is a client.

If your partition management program serves a termination request, it will be notified whenever any program on the workstation terminates. It can then identify whether or not the terminating program is its child, and take appropriate action. The general steps to do this are as follows:

1. The partition-managing program creates a child partition and loads a task into it.
2. Immediately after doing this, the partition-management program calls a procedure which waits for the child to terminate.
3. The termination procedure first disables Action-Finish and swapping, then serves the termination request.
4. When the termination procedure receives the termination request from the child user number, it unserves the request and returns to the main body of the program.
5. When the termination procedure returns, the main program knows that its child has terminated.

Note that the termination handler need not be just a procedure. It could be a separate process, which would allow the main program to continue running while the termination process waited for the child to terminate. The only caution is that whenever an application program serves a request, Action-Finish and swapping must be disabled.

Calling the Termination Procedure

Listing 4-4 shows a code fragment that calls the termination procedure, named WaitForTermination in the example.

```
erc = WaitForTermination(userNumChild);
if(erc == ercNoExitRunFile) {
  /* retrieve child's erc another way */
  erc = *(((Word *) pAscbChild) + 4);
  printf("\nChild partition exit run file not
       found.");
  }
```

Listing 4-4. Calling a Child-Termination Procedure

Structure of the Termination Procedure

Listing 4-5 shows an example of a child-termination procedure. Examine Listing 4-5 carefully, and you will see that it does more than simply check for a single termination request. The procedure must do this extra work in case the Context Manager software has been installed.

When the Context Manager is installed, multiple applications run at the same time. It is possible (though unlikely) for two programs to terminate almost simultaneously. Because of this possibility, programs which serve termination requests may have more than one outstanding request in their request queue.

Therefore, the termination procedure must empty its request queue before unserving its termination request.

```
ErcType WaitForTermination(Word userNum) {
  RqTerminationType *pRqBlocRet;
  Word iMessageType;
  FlagType fAlive = TRUE;
  ErcType erc, ercRet = ercNoExitRunFile;
static Word MyExchange = 0;
  rqInfoType rqOldExchange;
```

Listing 4-5. A Child-Termination Procedure (Page 1 of 2)

```
    RetPreCleanup(DisableActionFinish(0xFF));
    RetPreCleanup(SetSwapDisable(0xFF));

    if(MyExchange == 0)
        RetPreCleanup(AllocExch(&MyExchange));

    RetPreCleanup(QueryRequestInfo(lMyTerminationRq,
            &rqOldExchange, sizeof(rqInfoType)) );
    RetPreCleanup(ServeRq(lMyTerminationRq, MyExchange));

    while(fAlive != FALSE) {
        RetCleanup(Wait(MyExchange, &pRqBlocRet));
        /* if right user num to catch termination request */
        if(pRqBlocRet->rqHead.userNum == userNum)  {
            pRqBlocRet->rqHead.ercRet = ercOK;
            fAlive = FALSE;
            ercRet = pRqBlocRet->ercTermination;
            }
        Respond(pRqBlocRet);
        }

cleanup:
    while(erc != ercNoMessage) {
        /* make sure nobody's in queue */
        erc = Check(MyExchange, &pRqBlocRet);
        if(erc == ercOK)
            Respond(pRqBlocRet);
        }

    CheckErc(ServeRq(lMyTerminationRq,
            rqOldExchange.exch));
precleanup:
    CheckErc(SetSwapDisable(FALSE));
    CheckErc(DisableActionFinish(FALSE));
    return(ercRet);
}
```

Listing 4-5. A Child-Termination Procedure (Page 2 of 2)

Deallocating the Partition

Deallocating a partition is the simplest of the partition management operations. To deallocate a child partition, the parent program simply vacates the partition, then removes it. Listing 4-6 shows a procedure that performs these operations.

```
void KillPartition(Word userNum) {
  ErcType MyErc;

  MyErc = VacatePartition(userNum);
  if(MyErc != ercOK && MyErc != ercPartitionVacant)
    CheckErc(MyErc);
  MyErc = RemovePartition(userNum);
  if(MyErc != ercOK)
    CheckErc(MyErc);
}
```

Listing 4-6. A Partition Removal Procedure

A Sample Partition Management Program

Below is a complete listing of a simple partition management program. A sample request definition file for the program's termination request follows the program listing.

```
/*************************************
 * Filename:  Partitions.c
 * Date:      12/8/89
 * Author:    A. Coleman
 * Compiler:  Metaware C
 *
 * This file spawns a partition, runs a program in it, then kills the
 * partition.
 *
 *************************************/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define AllocExch
#define Check
#define CheckErc
#define CreateBigPartition
#define DisableActionFinish
#define ForwardRequest
#define GetPartitionStatus
#define GetpStructure
#define GetUCB
#define GetUserNumber
#define LoadInteractiveTask
#define LoadPrimaryTask
#define QueryBigMemAvail
#define QueryRequestInfo
#define RgParamInit
#define ServeRq
#define SetSwapDisable
#define SwapInContext
#define RemovePartition
#define Request
#define Respond
#define RgParamInit
#define RgParamSetSimple
#define VacatePartition
#define Wait
```

Listing 4-7. A Simple Partition-Managing Program (Page 1 of 7)

```
#define sdType
#define Syslit
#define UCBType
#define GetpStructureCase
#define RqHeaderType
#define rqInfoType

#include <ctoslib.h>
#define kernelErc
#define mpErc
#define RxErc
#define RqErc
#include <erc.h>

#define ParBackground 0
#define Par2          2
#define Par6          6
#define ParBatch      0xFF

#define StdPriority   0x80
#define lNotifyCMRq   258
#define lCMTerminationRq 273
#define lMyTerminationRq 0xE005
#define Go 0x1B

#define RetCleanup(x) if((erc = x) != ercOK) goto cleanup
#define RetPreCleanup(x) if((erc = x) != ercOK) goto precleanup

typedef struct {
   Word fhSwapFile;
   Pointer pVlpb;
   char unreferenced[58];
   char sbUserName[31];
   char sbPassword[13];
   char sbCmdFile[79];
} AscbType;
#define SCOPYAREA 31+13+79
/* size of sbUserName, sbPassword, sbCmdFile */

typedef struct {
   Byte level;
   Byte nLinesMax;
   Byte nColsNarrow;
   Byte nColsWide;
} VidHdwType;
```

Listing 4-7. A Simple Partition-Managing Program (Page 2 of 7)

```
typedef struct {
   RqHeaderType rqHead;
   ErcType      ercTermination;
   } RqTerminationType;

/* Just a status buffer */
Byte pStatus[250];

pragma Calling_convention(CTOS_CALLING_CONVENTIONS);
extern ErcType AltAllocMemoryLL(Word userNum, Word cBytes,
   Pointer ppSegmentRet);
extern ErcType AltInitCharMap(Word userNum, Pointer pMap, Word sMap);
extern ErcType AltInitVidFrame(Word userNum, Byte iFrame, Byte
   iColStart,Byte iLineStart, Byte nCols, Byte nLines, Byte borderDesc,
   Byte bBorderChar, Byte bBorderAttr, Byte fDblHigh, Byte fDblWide);
extern ErcType AltQueryVidHdw(Word userNum, Pointer pBuffer,
   Word sBuffer);
extern ErcType AltResetVideo(Word userNum, Byte nCols, Byte nLines,
   Byte fAttr, Byte bSpace, Pointer psMapRet);
extern ErcType AltSetNode(Word userNum, Pointer pbNodeName,
   Word cbNodeName);
extern ErcType AltSetPath(Word userNum, Pointer pbVolSpec,
   Word cbVolSpec, Pointer pbDirName, Word cbDirName,
   Pointer pbPassWord, Word cbPassWord);
extern ErcType AltSetPrefix(Word userNum, Pointer pbPrefix,
   Word cbPrefix);
extern ErcType AltSetScreenVidAttr(Word userNum, Word iAttr, Word fOn);
extern ErcType AltSetExitRunFile(Word userNum, Pointer ExitRunFile,
   Word cbRunFile, Pointer pbPassword, Word cbPassword, Word Priority);
pragma Calling_convention();

void KillPartition(Word userNum) {
   ErcType MyErc;

   MyErc = VacatePartition(userNum);
   if(MyErc != ercOK && MyErc != ercPartitionVacant)
      CheckErc(MyErc);
   MyErc = RemovePartition(userNum);
   if(MyErc != ercOK)
      CheckErc(MyErc);
}
```

Listing 4-7. A Simple Partition-Managing Program (Page 3 of 7)

```
/* NOTE:  This could be a separate process */
ErcType WaitForTermination(Word userNum) {
   RqTerminationType *pRqBlocRet;
   Word iMessageType;
   FlagType fAlive = TRUE;
   ErcType erc, ercRet = ercNoExitRunFile;
static Word MyExchange = 0;
   rqInfoType rqOldExchange;

   RetPreCleanup(DisableActionFinish(0xFF));
   RetPreCleanup(SetSwapDisable(0xFF));

   if(MyExchange == 0)
      RetPreCleanup(AllocExch(&MyExchange));

   RetPreCleanup(QueryRequestInfo(lMyTerminationRq, &rqOldExchange,
            sizeof(rqInfoType)) );
   RetPreCleanup(ServeRq(lMyTerminationRq, MyExchange));

   while(fAlive != FALSE) {
      RetCleanup(Wait(MyExchange, &pRqBlocRet));
      /* if is right user num and is termination request */
      if(pRqBlocRet->rqHead.userNum == userNum)  {
         pRqBlocRet->rqHead.ercRet = ercOK;
         fAlive = FALSE;
         ercRet = pRqBlocRet->ercTermination;
         }
      Respond(pRqBlocRet);
      }

cleanup:
   while(erc != ercNoMessage) {
      /* make sure nobody's in queue */
      erc = Check(MyExchange, &pRqBlocRet);
      if(erc == ercOK)
         Respond(pRqBlocRet);
      }

   CheckErc( ServeRq(lMyTerminationRq, rqOldExchange.exch));
precleanup:
   CheckErc(SetSwapDisable(FALSE));
   CheckErc(DisableActionFinish(FALSE));
   return(ercRet);
}
```

Listing 4-7. A Simple Partition-Managing Program (Page 4 of 7)

```
ErcType GetParams(Pointer pVlpb) {
#define MAXPARAMS 8
   ErcType erc;
   int     iCurrent = 1;
   sdType  sdParam;
   char    rgbString[80];

   sdParam.pb = rgbString;
   sdParam.cb = 0;

   erc = RgParamInit(pVlpb, 0, MAXPARAMS);
   printf("\nType param or press Go, then Return to execute.\n");

   while(TRUE) {
      printf("Enter parameter %d:  ", iCurrent);
      gets(rgbString);
      if(rgbString[0] == Go)
         break;
      sdParam.cb = strlen(rgbString);
      erc = RgParamSetSimple(iCurrent, ((Pointer) &sdParam));
      iCurrent++;
      }
   return(erc);
}

ErcType RunSubTask(char *pRunFileName, Word MemSize) {
   ErcType erc, ercRet=0;
   DWord   dMemAvail;
   Word    userNumChild, userNum;
   AscbType * pAscbChild, *pAscbMine;
   Pointer pVlpbMine, pVlpbChild;
   Pointer pTemp;
   UCBType ucb;
   VidHdwType vidHdw;
   Word sMap, cLines, cCols;

   erc = CreateBigPartition("MyChild", 7,(DWord) MemSize*64 /*parags*/,
              Par6, &userNumChild);
   if(erc != ercOK)
      return(erc);

   RetCleanup(SwapInContext(userNumChild));

   /* copy Vlpb and insert its address in the child partition's Ascb */
   RetCleanup(GetpStructure(lGetpVLPB, 0, &pVlpbMine));
```

Listing 4-7. A Simple Partition-Managing Program (Page 5 of 7)

```
RetCleanup(AltAllocMemoryLL(userNumChild,*(Word *)pVlpbMine,
         &pVlpbChild));
memcpy(pVlpbChild, pVlpbMine, *(Word *)pVlpbMine);
RetCleanup(GetpStructure(lGetpAscb, userNumChild, &pAscbChild));
pAscbChild->pVlpb = pVlpbChild;

/* copy user name, signon password, and sys.cmds spec in new Ascb */
RetCleanup(GetpStructure(lGetpAscb, 0, &pAscbMine));
memcpy(pAscbChild->sbUserName, pAscbMine->sbUserName, SCOPYAREA);

/* sneakily replace my pVlpb with childs so I can get params */
/* have to do this cuz RgParam ops are not requests, so no Alt
   versions available */
pTemp = pAscbMine->pVlpb;
pAscbMine->pVlpb = pVlpbChild;
/* get any parameters the user wants to enter */
RetCleanup(GetParams(pVlpbChild));
/* restore my Ascb */
pAscbMine->pVlpb = pTemp;

/* set child's exit run file */
RetCleanup(AltSetExitRunFile(userNumChild, "[Sys]<sys>Exec.run", 18,
         NULL, 0, 0x80));

/* copy current path to new partition */
RetCleanup(GetUCB(&ucb, sizeof(ucb)));
RetCleanup(AltSetPath(userNumChild, ucb.rgbVol, ucb.cbVol,
   ucb.rgbDir, ucb.cbDir, ucb.rgbPswd, ucb.cbPswd));
if (ucb.cbPrefix)
   RetCleanup(AltSetPrefix(userNumChild, ucb.rgbPrefix,
         ucb.cbPrefix));
if (ucb.cbNode)
   RetCleanup(AltSetNode(userNumChild, ucb.rgbNode, ucb.cbNode));

/* initialize exec screen - neither font double bar needed */
RetCleanup(AltQueryVidHdw(userNumChild, &vidHdw, sizeof(vidHdw)));
cLines = vidHdw.nLinesMax;
cCols  = vidHdw.nColsNarrow;
RetCleanup(AltResetVideo(userNumChild, cCols, cLines, TRUE, 0,
   &sMap));
RetCleanup(AltInitVidFrame(userNumChild, 1, 0, 0, cCols, 2,
   0, 0, 0, 0, 0));
RetCleanup(AltInitVidFrame(userNumChild, 2, 0, 2, cCols, 1,
   0, 0, 0, 0, 0));
RetCleanup(AltInitVidFrame(userNumChild, 0, 0, 3, cCols, cLines - 3,
   0, 0, 0, 0, 0));
```

Listing 4-7. A Simple Partition-Managing Program (Page 6 of 7)

```
   RetCleanup(AltInitCharMap(userNumChild, 0, sMap));
   RetCleanup(AltSetScreenVidAttr(userNumChild, 1, 0xFF));

   RetCleanup(LoadInteractiveTask(userNumChild, pRunFileName,
         strlen(pRunFileName), NULL, 0, StdPriority, FALSE));

   RetCleanup(SwapInContext(userNumChild));

   erc = WaitForTermination(userNumChild);
   if(erc == ercNoExitRunFile) {
      /* retrieve child's erc another way */
      erc = *(((Word *) pAscbChild) + 4);
      printf("\nChild partition exit run file not found or CM not
            installed.");
      }

cleanup:
   KillPartition(userNumChild);
   return(erc);
}

main() {
char rgbRunFile[80];
char rgbMemSize[20];
Word MemSize;
ErcType ercRet;

   while(TRUE) {
      printf("\nEnter run file name:  ");
      gets(rgbRunFile);
      if(rgbRunFile[0] == 0)
         break;
      printf("How much memory does it need (in K):  ");
      gets(rgbMemSize);
      MemSize = atoi(rgbMemSize);
      ercRet = RunSubTask(rgbRunFile, MemSize);
      printf("\nExecuted %s.  Returned erc = %d\n", rgbRunFile,
            ercRet);
      }
   return(0);
}
```

Listing 4-7. A Simple Partition-Managing Program (Page 7 of 7)

```
:WsAbortRq:      0E005h
:TerminationRq:  0E005h

:RequestCode:    0E005h
:RequestName:    TerminationAndWsAbortRq
:Version:        1
:LclSvcCode:     0002h
:ServiceExch:    exchInstalledMastr
:sCntInfo:       6
:nReqPbCb:       0
:nRespPbcb:      0
:Params:         none
:NetRouting:     noRouting
:SrpRouting:     rLocal
```

Listing 4-8. A Termination Request Definiton File

5

Software Installation: The Installation Manager

Introduction

If you have written an application or system service, users need to install it as a software product on the hard disk of their workstation or server. You need to create several files to accomplish this process. This chapter describes the installation process and provides guidelines for writing the installation files as well as instructions on how to place these files on the installation media. This chapter assumes that you are familiar with the batch processing of job control language (JCL) files. If you are not, refer to the *CTOS Batch Manager II Installation, Configuration, and Programming Guide* before proceeding.

Two run files, two message files, and a command file provided with Standard Software, Version 12.1, work together to install software. These files are *InstallMgr.run, StdSoftMsg.bin, Batch.run*, *BatchMsg.bin*, and *Install>English.cmds*.

InstallMgr.run initiates installation and deinstallation of software. It does the preliminary work needed to accomplish the installation, such as collecting installation parameter data from the user configuration file or interactively from the user, verifying the availability of resources required by the product, and copying installation files to a temporary location on the hard disk where their contents can be accessed by *Batch.run*. *English>Install.cmds* contains commands commonly used in installation scripts. Batch uses this file to resolve $Command statements in the installation scripts.

Once the installation environment is set up, the Installation Manager calls the Chain operation to pass control to Batch. Batch does the actual work of installing your software product. Guided by information provided by the Installation Manager, Batch performs the job steps in the installation script. These steps typically include copying the files for your product to

the hard disk and merging your product's commands with the proper command file. For your understanding of the installation process, this chapter begins by describing the Installation Manager and Batch programs in greater detail. Following the program descriptions, this section describes the contents of the installation files, explains how you create these files, and instructs you on how to organize the content of your distribution media. Later in this section, example scripts show you how to write the instructions Batch needs to carry out the installation.

Key Concepts

This section describes key concepts related to the installation process: the type of installation (floppy, tape, or server), whether the installation is public or private, and whether the product needs to be divided into subpackages for separate installation.

Types of Installation: Floppy, Tape, Server

The *installation media* is the type of media on which your product is sold to the customer. It could be 5 1/4-inch or 3 1/2-inch disks or Quarter-Inch Cartridge (QIC) tape. A *floppy installation* is an installation whose media is floppy disks. A *tape installation* is an installation whose media is QIC tape.

A *server installation* is an installation whose media is a disk on the server. A subpackage must be installed *publicly* before it can be installed from the server.

Public vs. Private Installation

A *public* installation is a floppy or tape installation that puts files onto the server and commands into the file *[!Sys]<Sys>Cluster.cmds*. When a subpackage is installed publicly, it is made available to all members of the cluster without having to be installed on the cluster workstation's disk. A number of special fields must be included in the control file (see below) when a subpackage is installed publicly. These fields are omitted if the installation is private. A *private* installation puts files onto a hard disk on the workstation from which the Installation Manager is invoked.

Subpackages

A *product* is the set of files that make up the entity you are distributing. It consists of system services, (client) run files, and supporting files (for example, configuration files and request files) required by your service and/or run files. These are files that you create and distribute. Your product does not include files like the Context Manager's configuration file, which is part of the Context Manager product, nor *Request.sys*, which is part of the Standard Software product.

Before putting your product onto distribution media, you must decide whether it is atomic or can be divided into smaller pieces. This is determined most of the time by answering the question: "Are there parts of the product that some users would want and others would not?" For example, Standard Software consists of more than 15 pieces (subpackages). On a smaller scale, a product might consist of a system service (that would only be installed on the server) and several client programs. In this case, you would divide it into two subpackages. Using subpackages allows you to take full advantage of the variables and other resources that the Installation Manager provides to you as script author.

A *subpackage* is a part of an application that is self-sufficient. It will have its own entry in the installation database. For instance, an accounting product might have four subpackages: Accounts Payable, Accounts Receivable, Payroll, and General Ledger.

The *installation database* is the file where the Installation Manager keeps the information about the subpackage. It gets the information from the control file, described later in this chapter. Subpackages that are installed publicly are put into *[!Sys]<Installed>Cluster.installed*. Subpackages that are installed locally are put into *[Sys]<Installed>username.installed*.

A *package* is a special case where the product consists of one subpackage.

The Installation Manager

To install a software product, a user invokes the Installation Manager using the Executive **Installation Manager** command. Information on using this command is contained in the *CTOS System Administration Guide*. The Installation Manager determines the type of installation the user wants to perform and sets up the appropriate environment.

A user can also specify directly which type of installation is needed through use of the **Floppy Install, Tape Install,** or **Server Install** commands.

When invoked by the **Installation Manager** command, the Installation Manager performs the following procedure sequence. When invoked by the **Floppy Install, Tape Install,** or **Server Install** commands, the sequence begins at step 3.

1. It allows the user to choose a function – **Install New Software, Show Installed Software,** or **Remove Installed Software.** We will assume the user chooses the **Install New Software** option.

2. It queries the user for the source of the installation, such as floppy disk, QIC tape, or the server.

3. It sets the installation variables based on parameter values obtained from the user configuration file (or from the user choosing the **Examine/Change Defaults** option from the Installation Defaults menu). (All the installation variables are described in "Installation Variables," later in this chapter.)

4. It then asks the user to choose the subpackage(s) s/he wishes to install. If the installation is performed from the server, only one subpackage may be chosen at a time.

5. It calculates whether disk space is sufficient for installing the product using data obtained from the installation control file (described later).

6. The Installation Manager then chains to Batch and passes the values of the installation variables to Batch.

7. Batch processes the installation script and then returns control to the Installation Manager.

8. If the installation succeeds, the database will be updated to reflect the installation of the subpackage.

If the installation type was from floppy and the user selected more than one subpackage, steps 6 through 8 are repeated for each subpackage.

If the installation type was from server, then the user begins again at step 1. Note that step 3 only has to be done once each time the Installation Manager is invoked. The Installation Manager does not (re)set these values except upon initial invocation and during step 3.

There is only one installation script for tape installation, regardless of the number of subpackages that the user selects. Therefore steps 6 and 7 are only done once, but step 8 is performed once for each subpackage selected.

Batch

Once the Installation Manager passes control to Batch, Batch does the actual work of installing your subpackage on the hard disk. Using the values in the installation variables, Batch executes the statements in your installation script.

Instructions in the installation script typically include:

- copying the files for your subpackage from the installation media to the appropriate destination on the hard disk.
- merging your requests into the user's request file.
- merging your commands into the appropriate command file.

See the end of the chapter for examples of installation scripts.

Installation Files

You create four installation files as part of your subpackage:

- a control file (required), *Install.ctrl*
- an installation script file (required), *Install.jcl*
- a message file (optional), *InstallMsg.bin*
- a command file (optional), *Install.cmds*

The Installation Manger reads these files from the installation media and copies them to a temporary location on the hard disk where their contents are available for the duration of the installation process. The first two files are ASCII files and are created by using the Editor. The message file is created from an ASCII file using the Executive command *Create Message File*. The command file is a typical command file and is created by the Command File Editor.

Control File

The *control file* contains information needed to guide the installation process. The Installation Manager reads the contents of this file and uses the information it contains to perform functions, such as

- displaying the package name and version number on the user screen
- verifying that there are adequate space and resources for the installation
- updating the database

All information in the control file is placed in the user's installation database upon successful installation.

The control file for Mouse Services, for example, contains the following fields and parameter entries:

```
:PackageName:Mouse
:Version:12.1.0
:RequiredDiskSpace:381
:Verify:
:Directory:<Sys>
:Commands: 'Install Mouse Service' 'Deinstall Mouse Service' 'Set
Mouse Controls'
:ProductFiles:       1024x768_PntSys.icon       720x348_PntSys.icon
DefaultCursor.icon Mouse.run MouseBm.run MouseCh.run MouseForm.lib
MouseMsg.bin SetMouse.run
```

Values for each keyword start on the same line as the keyword. Spaces may exist between the ending colon of the keyword and the beginning character of the value. The value is terminated by a carriage return. This is especially important to remember when dealing with values such as *:ProductFiles:* which contain a list of files that might wrap around several

lines. Do not use carriage returns to make the file look pretty. A carriage return *terminates* the list. If the value is a list of different elements, separate the elements with a space, not a comma. Place single quotation marks around individual elements if they contain embedded spaces.

Entries are optional unless so stated. The keywords and a description of each field are described below.

The contents of this file are of the form :keyword: value

PackageName (required)

The *PackageName* field should contain a name that clearly identifies the software product to the user. The maximum length is 30 characters. Embedded spaces are allowed. This name must be unique. "OFIS Designer" is a good name, whereas its part number would not be a good name. (Note that the field name should technically be *SubPackageName*, but remains *PackageName* so old installations do not break.)

MultiPkgName (optional)

The *MultiPkgName* specifies the name of a subpackage. See *PackageName* above for restrictions. If this entry appears in a control file, then the control file must contain exactly one *PackageName* entry and between two and fifteen *MultiPkgName* entries, and nothing else.

Version (required)

The *Version* field lets the Installation Manager know if the same or an earlier version of the product is already installed, or if the installation is being performed for the first time. The maximum length is 15 characters. Embedded spaces are not allowed.

Commands (optional)

The *Commands* field contains a list of all commands used to invoke the run files and/or services in the subpackage. These commands will be removed from the command file when the user removes the subpackage.

Directory (optional)

The *Directory* field specifies the directory in which the files will reside. Include angle brackets. Usually the value is <Sys>. The maximum length is 14. If you use this entry, do not put the directory in the file specifications given in the *ProductFiles* and *OverwriteNoFiles* fields. If you must place files in more than one directory, do not use this entry. Instead, place the directories in the file specifications given in the *ProductFiles* and *OverwriteNoFiles* fields.

ProductFiles (optional)

The *ProductFiles* field includes a comprehensive list of files that make up your subpackage. They are deleted when the subpackage is deinstalled. When the user requests a backup of the previous version of software (the next time you release your product), these files along with those listed in the *RelatedFiles* field will be archived. The length of these files should be added and that number placed in the *RequiredDiskSpace* entry. Do not include volume information in the file name. The user controls the volume name. Do not include directory information if you included the *Directory* field. The directory name will be taken from the *Directory* field.

If the verify feature is enabled, after returning from Batch, the Installation Manager checks the files in the *ProductFiles* list to see if their modification/creation date-time stamps are greater than when the Installation Manager was invoked. If they are not, the installation fails and a message with the file name of the non-updated file is put in the installation log file.

PublicProductFiles (optional)

The *PublicProductFiles* field contains a list of files that make up your subpackage when it is installed publicly. If there is no difference in a public and private installation, do not include this entry. If some files are common to both public and private installations, make sure you include them in both the *PublicProductFiles* and *ProductFiles* fields.

If the verify feature is enabled, after returning from Batch, the Installation Manager checks the files in the *PublicProductFiles* list to see if their modification/creation date-time stamps are greater than when the Installation Manager was invoked. If they are not, the installation

fails and a message with the file name of the non-updated file is put in the installation log file.

OverwriteNoFiles (optional)

The *OverwriteNoFiles* field specifies a list of file names that belong to your subpackage, but which should not be overwritten by the installation process if they already exist. Do not include volume information in the file name. The user controls the volume name. Do not include directory information if you included the *Directory* field. The directory name will be taken from the *Directory* field. User-configurable files would go into this list–for example, *Queue.index*. These names must also be in the *ProductFiles* list.

If the verify feature is enabled, after returning from Batch, the Installation Manager checks the files in the *OverwriteNoFiles* list to see if they exist. If they don't, the installation fails and a message with the file name of the nonexistent file is put in the installation log file. Remember to include these file names in the *ProductFiles* list.

PublicOverwriteNoFiles (optional)

The *PublicOverwriteNoFiles* field contains a list of files that belong to your subpackage, but which should not be overwritten by a public installation if they already exist. See *OverwriteNoFiles* for restrictions. The names here must also be in the *PublicProductFiles* list. If there is no difference in a public and private installation, do not include this entry.

If the verify feature is enabled, after returning from Batch, the Installation Manager checks the files in the *PublicOverwriteNoFiles* list to see if they exist. If they don't, the installation fails and a message with the file name of the nonexistent file is put in the installation log file. Remember to include these file names in the *PublicProductFiles* list.

RelatedFiles (optional)

The *RelatedFiles* field is a list of files that the installation process will update but which are not part of your product–for example, *Request.sys*. At this time there is no way to specify files such as the *.user* file and the Context Manager's configuration file, since the names can be changed by the user at install time. These files are archived along with the files listed in the *ProductFiles* entry.

RequiredDiskSpace (optional)

The *RequiredDiskSpace* field specifies the total length (in K Bytes) of all installation files in your subpackage. Although the units are kilobytes, do not append "K" to the number. (See also *ProductFiles* above.)

PublicRequiredDiskSpace (optional)

The *PublicRequiredDiskSpace* field is the total size of all the files in your subpackage when it is installed publicly. See *RequiredDiskSpace* for restrictions. If there is no difference in a public and private installation, do not include this entry.

RequestCodes (optional)

The *RequestCodes* field is a list of request codes for which your subpackage issues ServeRq calls. The codes must be separated by a space, not a comma. The request codes can be either decimal or hexadecimal numbers. If they are hexadecimal, don't forget to append "h," for example, 0E001h. These request codes will be removed when your subpackage is deinstalled.

RequiredMemory (optional)

The *RequiredMemory* field specifies the amount of memory your largest run file requires. This entry is not currently used.

Verify (optional)

The *Verify* field, if included, indicates that the Installation Manager should verify the time/date stamp during installation. This field contains no value. This entry should always be included since the verify feature is a useful function of the Installation Manager. If only a portion of the product files is to be installed, dividing the product into appropriate subpackages allows the user to install an entire subpackage, and to verify the time and date of each product file.

NoDbUpdate (optional)

The *NoDbUpdate* field, if included, indicates that the Installation Manager should not update the installation database. Normally this field would not be included in the control file. However, some remote installations (via SWD, a different CTOS installation method) may want

only to copy files or perform some other functions and therefore don't want the database updated.

Installation Script File

The *installation script file* contains Batch JCL instructions that will cause the installation of a subpackage. You are responsible for writing the contents of this file, without which an installation will fail. Because most installation errors occur when Batch executes the script file, you must pay careful attention to writing the scripts.

Your installation can also use multiple JCL files. Your main JCL file can execute other JCL files by using the **$Call** command. Using the **$Call** command makes your installation scripts more modular. This can ease script development and speed up the user's software installation process. See your Batch documentation for more information on nesting JCL files. Using more than one JCL file is not recommended, however, since the subpackage concept and use of the installation variables and the message file greatly reduce the amount of work that a JCL file has to do. If you do use multiple JCL files, you are responsible for making them accessible to Batch for all types of installations, especially installations from the server. Also, the additional JCL files will not be deleted when a package is removed (the main JCL file, message file, and command file are deleted because the Installation Manager knows their names).

Since you should allow the user to install your subpackage publicly, the script file(s) must handle installation from server as well as installation from the media (floppy or tape).

Message File

In order to aid nationalization efforts, you should keep all nationalizable text strings in a *message file*. In general, only two nationalizable strings will be needed in the script file: one that is used to report the absence of the message file and one to be used as an argument to NextFloppy() to ensure that the correct disk is mounted. (By ensuring that the correct disk is mounted, the user should never see the message reporting the absence of the message file, since it will be on the disk that should be mounted.) After creating the message file, use the Executive command **Create Message File** to create the binary message file that will be placed on the distribution media.

Command File

The *command file* for a software product contains all the commands that need to be added to the system in support of that product. Each product command file is a standard command file like the Executive command file, *Sys.cmds*, but usually with fewer entries in it. Don't forget to place all the names in the *Commands* field of the control file.

Use the **Command File Editor** Executive command to create the file containing the commands that will be used to invoke your run file(s) and/or install your system service. You should use the **Merge Command Files** command statement in your script file to update the user's command file. We recommend that you do not use **New Command** in the script file.

NOTE: The command file is required only if your product has associated commands.

The command file for Mouse Services, for example, contains the following entries:

Install Mouse Service
Deinstall Mouse Service
Set Mouse Controls

For more details on how to create a customized command file, see the description of the **Command File Editor** in the *CTOS Executive Reference Manual*.

The Installation Manager does not read the contents of a command file. It merely copies the file to a temporary location so that the file contents are available to Batch. During the installation process, your script file, using the installation variables CmdFileFrom, CmdFileTo, and VolumeTo, tells the Command File Editor to merge the commands in this file with the user-specified command file.

Because a command file merge adds all the commands to the user-specified command file, you have the option of creating several command files for your product. By doing so, each command file you create can be merged with the command file of the user. This allows you

to selectively merge only those commands the user may need. However, if you use more than one command file, your script is responsible for placing it somewhere where it will be available to all types of installations, including installation from server.

User Configuration File

In addition to reading the installation files just described, the Installation Manager reads the contents of the user configuration (*.user*) file. It then sets the following values of the installation variables:

- installation type (public or private)
- video level (verbose or silent)
- whether to archive the currently installed version of the subpackage
- whether to save the archive
- the volume and directory into which to create the archive (if archiving is desired) when the installation is private
- the volume and directory into which to create the archive (if archiving is desired) when the installation is public
- the volume upon which to install the software when the installation is private
- the volume upon which to install the software when the installation is public
- The name of the Context Manager configuration file to use when the installation is private
- The name of the Context Manager configuration file to use when the installation is public
- Whether or not to use a log file
- The name of the log file

As a script writer, you do not have to be concerned about how the user configuration file (or the user interactively) influences an installation.

There is one parameter, however, that you need to take into account: that is, whether the installation is public or private. You only need to be aware of this value if the distribution media is floppy disks and the files for a public installation are not the same as a private installation. In all other cases, by using the installation variables provided by the Installation Manager to Batch, this attribute is transparent to you.

Naming Your Floppy Installation Files

In the simplest case, the product is atomic. It has one name in the database. In this case, the names of the installation files are

Install.ctrl
Install.jcl
InstallMsg.bin
Install.cmds

With subpackages, you have at least two levels of control files. The top level is always called *Install.ctrl*. To illustrate, let's use the example of an accounting package, Accounts for You, pictured in Figure 5-1.

The *Install.ctrl* file for Accounts for You would have the following entries:

:PackageName: Accounts for You
:MultiPkgName: Accounts Payable
:MultiPkgName: Accounts Receivable
:MultiPkgName: Payroll
:MultiPkgName: General Ledger

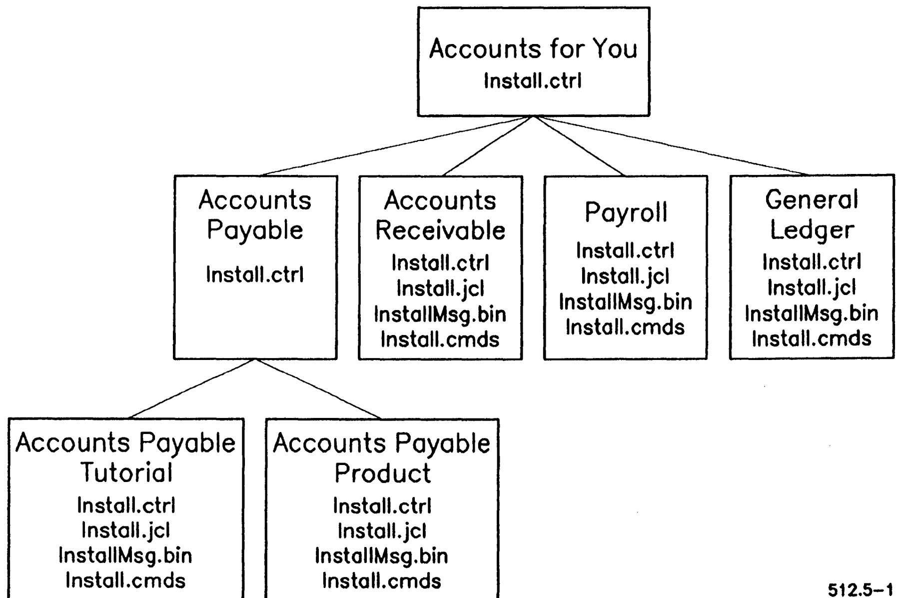


Figure 5-1. Files used in a floppy installation for an accounting package

The Accounts Receivable subpackage has its own control file, installation script, message file, and command files, as shown in the figure. So do the Payroll and General Ledger subpackages. They are called

AccountsReceivable>Install.ctrl
AccountsReceivable>Install.jcl
AccountsReceivable>InstallMsg.bin
AccountsReceivable>Install.cmds
Payroll>Install.ctrl
Payroll>Install.jcl
Payroll>InstallMsg.bin
Payroll>Install.cmds
GeneralLedger>Install.ctrl
GeneralLedger>Install.jcl
GeneralLedger>InstallMsg.bin
GeneralLedger>Install.cmds

The Accounts Payable subpackage is further subdivided into two subpackages: Accounts Payable Tutorial and Accounts Payable Product.

Accounts Payable has only a control file, since it is not yet at the bottom of the product "tree." Its two subpackages each have all four installation files, since they are at the bottom of the tree. The *AccountsPayable>Install.ctrl* file would have the following entries:

:PackageName:Accounts Payable
:MultiPkgName: Accounts Payable Tutorial
:MultiPkgName:Accounts Payable Product

In this example, there are two layers before the user actually chooses a subpackage. The user is presented a menu containing "Accounts Payable," "Accounts Receivable," "Payroll," and "General Ledger." After Accounts Payable is chosen, another menu with the contents of *AccountsPayable>Install.ctrl* is displayed and the user chooses which of those subpackages to install.

During installation, the Installation Manager reads *Install.ctrl* and displays a menu. When the user makes a selection (let's say that all five items are chosen), the Installation Manager will try to open the control file related to the first (top) entry, *AccountsPayableTutorial>Install.ctrl*. If this file is not found, it will prompt the user to mount the proper disk. If it is found, the installation files (*AccountsPayableTutorial>Install**) are copied to [Scr]<$>. The control file is read and information is extracted. Note that the *:PackageName:* entry must match the prefix part of the name of the control file. Control is then passed to Batch, which will execute *AccountsPayableTutorial>Install.jcl*. When the first subpackage has been successfully installed, the Installation Manager then tries to read *AccountsPayableProduct>Install.ctrl*, and the process repeats itself.

NOTE: The Installation Manager will make all the above files, but no others, available to Batch by copying them to [Scr]<$>. If the installation is public, the Installation Manager will also copy them to [!Sys]<Installed>PkgName>Version>Install, which makes them available when a user installs the package from the server. If your script calls other scripts or uses other command files, or uses any other files, it is your responsibility to make the script copy them from the floppy to where they will be accessible for later use.*

Naming Your Tape Installation Files

The method of installing a product from tape is slightly different than a floppy installation and therefore requires a slightly different scheme.

Tape installation requires all of the above control files and command files. However, it only uses one message file and one script file, *InstallMsg.bin* and *Install.jcl*. The reason is that there is one message file for every script file, and there is only one script file, *Install.jcl*. Figure 5-2 shows the installation files required for a tape installation for the same product. All 24 installation files should be archived to [Qic]0.1

Note that *Install.jcl* must do the work of installing all the subpackages that the user might choose. It will need to query the installation variable Pkgs to determine the packages that the user has chosen. The individual installation control files are used just as they are in a floppy installation. The individual installation script, message, and command files (for example, *Payroll>Install.jcl,* *Payroll>InstallMsg.bin,* and *Payroll>Install.cmds*) are not used by the tape installation but are included in case the user chooses to do a public installation. In this case, the script, message, and command files are copied to *[!Sys]<Installed>* for subsequent use when someone does an installation from server.

NOTE: The Installation Manager will put all the files in the Installation Archive File ([Qic]0) into [Scr]<$>. If your script uses other files, it is your responsibility to have the script move them to somewhere permanent so they are accessible when the user installs your product from the server.

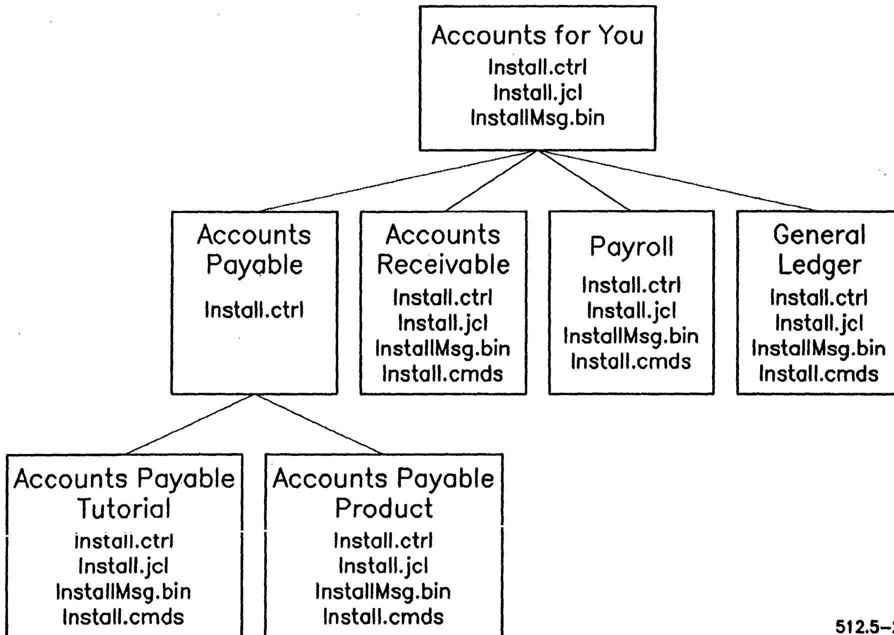


Figure 5-2. Files used in a tape installation for an accounting package

Organizing Your Installation Media

Once you have created the files you need for your program and its installation, you place these files on the appropriate distribution media.

For a Floppy Installation

For floppy disks, the control file, installation script, message file, and command file must reside in the <Sys> directory of the floppy disk. If your product contains only one subpackage, the installation files go onto the first disk. However, if you have more than one subpackage, then the *Install.ctrl* file goes on the first disk, and the installation files go on the first disk that contains the product files associated with that subpackage. For example, say that a product contains three subpackages, a word processor, a spell checker, and a grammar checker.

Disk Number	*Subpackage Name*	*Installation Files*
1	Word Processor	*Install.ctrl*
		WordProcessor>Install.ctrl
		WordProcessor>Install.jcl
		WordProcessor>Install.cmds
		WordProcessor>InstallMsg.bin
		SpellChecker>Install.ctrl
		SpellChecker>Install.jcl
		SpellChecker>Install.cmds
		SpellChecker>InstallMsg.bin
		GrammarChecker>Install.ctrl
		GrammarChecker>Install.jcl
		GrammarChecker>Install.cmds
		GrammarChecker>InstallMsg.bin
2	Word Processor	
3	Word Processor	
4	Word Processor and Spell Checker	*SpellChecker>Install.ctrl*
		SpellChecker>Install.jcl
		SpellChecker>Install.cmds
		SpellChecker>InstallMsg.bin
5	Spell Checker	
6	Spell Checker	
7	Grammar Checker	*GrammarChecker>Install.ctrl*
		GrammarChecker>Install.jcl
		GrammarChecker>Install.cmds
		GrammarChecker>InstallMsg.bin

In the case where two subpackages fit on one disk, all the installation files for those subpackages must be placed on that disk.

It is convenient for the user if all installation files are duplicated on the first disk, although this is not required. If this is done, the user has a better idea of what disk to insert. For example, let's say that the user inserted the first disk and invoked the Installation Manager. A menu would be presented allowing him to choose the Word Processor, Spell Checker, or Grammar Checker. Let's say the third subpackage, Grammar Checker, is chosen. The Installation Manager would try to open *GrammarChecker>Install.ctrl*. Since it is not on the first disk, the Installation Manager would prompt the user with a message like "Please insert the first disk of the Grammar Checker software." The user would then have to look in the release notice to find out which of the 7 disks

was the first disk of the Grammar Checker. However, if all the installation files are placed on disk 1 (and on their proper other disks as described above) then *GrammarChecker>Install.ctrl* would be found, the files would get copied to [Scr]<$>, and Batch would be invoked. The first statement in the installation script should always be a NextFloppy statement for the proper disk, in this case: NextFloppy('WordsforYou 7 of 7'). Now the user knows exactly which disk to insert.

Note that even when all the installation files are placed on the first disk, the installation files for the Spell Checker must still go on disk 4 and those for the Grammar Checker onto disk 7. The reason is that the user may choose more than one subpackage. Only the installation files for the one subpackage get copied to [Scr]<$> at a time. So if the user chooses all three subpackages, WordProcessor>Install* is copied to [Scr]<$>, then control is passed to Batch, then the Installation Manager updates the database and then tries to open SpellChecker>Install.ctrl. Since the user has disk 4 mounted, it is logical to put it there instead of making the user reinsert disk 1 just to get the installation files.

For a Tape Installation

For tape the installation files should be placed in [Qic]0. By placing all your product files on the next tape file, (in this case, [Qic]1), you will be able to take advantage of the DeviceFrom variable in your script file. In general, even numbered tape files should contain installation files and the next tape file should contain product files. However, as we will show later, even the most complicated tape installation can be easily done by using only [Qic]0 and [Qic]1.

After the user has had the opportunity to change the installation variable values, the Installation Manager uses *RestoreArchive.run* to place all the files from the Installation Archive File (usually [Qic]0) into [Scr]<$>. The Installation Manager reads the control files as necessitated by the user's choices. It builds lists of files that belong to the chosen subpackages and puts them in files in [Scr]<$> where they are available to the script. It then passes control to Batch, setting *Install.jcl* as the script file and *InstallMsg.bin* as the message file. The script file then should then perform the actual installation. When the Installation Manager is reloaded, it checks that all subpackages were installed correctly and updates the database for each subpackage.

Installation Variables

The *CTOS Batch Manager II Installation and Configuration Guide* contains most of the following information. It is repeated here with additional information relating specifically to the installation process. These installation variables are set by the user in the Installation Manager before the installation script is processed by Batch. They should not be changed by the installation script. Since the values are passed to Batch by the Installation Manager, the values are only valid when Batch has been invoked by the Installation Manager. Since all these variables can be set by the user, you must use them if they apply to your installation. For example, you cannot assume that if an installation is public, that your files are being copied to [!Sys]; You must use the variable VolumeTo. Likewise, you cannot assume that for private installations your commands should be merged into [Sys]<Sys>Sys.cmds. You must use CmdFileTo.

These "variables" are read-only. Although Batch will not report an error if you assign a value to them (for example, VolumeTo = '[Sys]'), this approach may thwart the desires of the user and is not recommended.

Variable Name	**Description**
CmdFileFrom	is the name of the command file that contains the commands for the subpackage. It is set to *subpackage>Install.cmds*. It should be used as the argument to the *Command File From* parameter in the **Merge Command Files** command.
CmdFileTo	is the name of the command file into which the commands in *CmdFileFrom* will be merged. The name is set by the user during the Examine/Change defaults phase of the installation. However, if the installation is public, the value is always *[!Sys]<Sys>Cluster.cmds*. It should be used as the argument to the *Command File To* parameter in the **Merge Command Files** command.

DeviceFrom

for floppy installations (both 5¼" and 3½"), this variable is the disk drive in which the installation disk is mounted. It is of the form [f*n*] where *n* is the drive number. For tape installations, this variable is the tape specification. The value is one more than the tape specification given by the user. In other words, if the user did not change the tape specification from its default of [Qic]0, its value would be [Qic]1, the tape file where your product files are archived. It should be used as the argument to the *Archive Dataset (QIC)* parameter of **Restore Archive**. For server installations, it is the node name ({Master}) and volume name of the volume to which it was publicly installed. (The Installation Manager keeps track of it in the database).

CmConfigFile

is the name of the user's Context Manager Configuration File. The name is set by the user during the Examine/Change defaults phase of the installation. It should be used as the argument to the *CM Config File* parameter in **CM Add Application.** The default is *[Sys]<Sys>CmConfig.sys.*

DirectoryTo

is the directory name, including angle brackets, into which the product files will be copied. If your product files must be copied into more than one directory, you should disregard this variable. It will always be <Sys> for tape installations.

InstallType

is the type of installation that the user is performing:

0 = Floppy
1 = Tape
2 = Server

MsgFile

is the name of the message file associated with the subpackage. It is meant to be used as an argument to the Batch function InitMsgFile. For tape installations it will always be *InstallMsg.bin*.

Pkgs

is a string containing the subpackage currently being installed and all subsequent subpackages that the user has chosen. Calls to the Batch function SubString should be used to query the contents of this variable. If other subpackages have been chosen, you can prompt the user to insert the correct diskette to continue the installation.

Public

is a flag that is set to TRUE if the user has chosen a public installation. Otherwise, it is set to FALSE. Note that the proper way to interrogate a flag in an if statement is:

```
If Public
...
EndIf
```

DO NOT USE

```
True = 65535
If Public = True
...
Endif
```

because you may set the value of True incorrectly and it just takes up room in Batch's symbol table.

Unattended

is a flag used to indicate that the user does not want to be asked any questions during the installation script. This variable should always be queried before performing the Batch functions DisplayAndWait, UserEnterValue, UserSelectMultiple, UserSelectSingle, and UserSelectYesNo. It

will always be FALSE during a floppy installation.

VolumeTo

is the volume name, including brackets, of the volume onto which the user wants the subpackage(s) installed. It should also be used as the argument to the *Default Volume* parameter in the **Merge Command Files** command.

File Lists

The Installation Manager also creates four at-files that are extremely useful for tape and server installation.

[Scr]<$>InstallOverwriteNoFrom.fls

This is a list of file names that were listed in the control file entry *OverwriteNoFiles* (or *PublicOverwriteNoFiles* if the installation is public).

For floppy and server installations, this list contains only files in the subpackage being currently installed. The file specifications contain volume, directory, and file name. For floppy installations, the volume is the same as *DeviceFrom* and the directory is the same as *DirectoryTo*. Since *DirectoryTo* is usually <Sys>, and the product files are usually (and should be) placed in a directory other than <Sys>, the list is not very useful for floppy installations. For server installations, the values for volume and directory are taken from the installation database.

For tape installations it contains files in every subpackage selected by the user. The file specifications are of the form <*>filename, making the list suitable as an argument to the [File list from (<*>*)] parameter of the Executive command **Restore Archive**.

[Scr]<$>InstallOverwriteNoTo.fls

This is a list of file names that were listed in the control file entry *OverwriteNoFiles* (or *PublicOverwriteNoFiles* if the installation is public). The list contains the same file names as *InstallOverwriteNoFrom.fls*, but the volume and directory information are different.

For floppy and server installations, this list contains only files in the subpackage being currently installed. The file specifications contain volume, directory, and file name. The volume is the same as *VolumeTo* and the directory is the same as *DirectoryTo*.

For tape installations it contains files in every subpackage selected by the user. The file specifications contain volume, directory, and file name. The volume is the same as *VolumeTo* and the directory is the same as *DirectoryTo*, making the list suitable as an argument to the [File list to (<*>*)] parameter of the Executive command **Restore Archive**.

[Scr]<$>InstallOverwriteOkFrom.fls

This is a list of file names that were listed in the control file entry *ProductFiles* (or *PublicProductFiles* if the installation is public). See the description for *InstallOverwriteNoFrom.fls*, earlier.

[Scr]<$>InstallOverwriteOkTo.fls

This is a list of file names that were listed in the control file entry *ProductFiles* (or *PublicProductFiles* if the installation is public). See the description for *InstallOverwriteNoTo.fls*, earlier.

Restarting an Installation

The sample installation scripts in the following section contain a number of restart labels ("RestartLabel"). This section includes a brief description of how Batch and the Installation Manager interact with respect to restart labels, which are markers within the JCL file that indicate points where processing of the JCL file can be restarted.

Every time Batch encounters a restart label, it opens the JCL file header and writes the number of restart label statements it has encountered. After each nonintrinsic statement Batch encounters (such as a command statement or a run statement), it looks in the ASCB to see if the application completed successfully. If it did not complete successfully (and *ContinueOnError* was not specified), then Batch exits and loads the Installation Manager. The Installation Manager looks in the ASCB for a nonzero status code. It then opens the JCL file that was being processed and looks at the file header to see if it can be restarted (that is, the number of restart labels is greater than zero). The Installation Manager

puts some additional information into the file header, reports the error to the user, and exits.

The user looks in the log file and fixes the problem (for example, if the directory was full, creating a bigger directory or deleting some files might help).

Then the user invokes the Installation Manager again. The Installation Manager looks for JCL files, opens the file and checks the file header to see whether it can be restarted. If it can be restarted, the Installation Manager tells the user if the last installation failed and allows him to restart it.

If the user specifies to restart the previous installation, control passes to Batch, which opens the file and looks for the last restart label it encountered. It then begins processing the JCL file from that point onward.

Note that the mere presence of restart labels in a JCL file does not guarantee that the installation can be restarted in all cases. First, the error must have occurred after Batch has read at least one RestartLabel statement in the JCL file. Second, the utility reporting the error must place the status code in the ASCB (otherwise the Installation Manager will have no way of knowing that an error occurred). Third, Batch must have time to write the necessary information in the file header of the JCL file before it returns control to the Installation Manager.

Nationalization

The file *[Sys]<Sys>Install>English.cmds* contains common commands (in English) used in installation scripts. This file is provided so that the $Command statements used in installation scripts do not have to be translated from English into the local language.

Check to be sure that all the command names that you use with $Command statements are contained in *[Sys]<Sys>Install>English.cmds.* If a command used in your script is not in this file, add the command through use of the **Merge Command Files** command or the **New Command** command (in your script file). Do not overwrite the existing *Install>English.cmds* file, because other products might rely on commands they have added to it.

The commands that are in the 12.1 version of *Install>English.cmds* are Append, CM Add Application, CM Remove Application, Copy, Create Directory, Create File, Create Message File, Delete, Expand File, Format Disk, Install New Requests, Install Sequential Access Service, Install Xbif Service, LCopy, List Request Set, Make Request Set, Merge Command Files, Move, New Command, Path, Remove Command, Remove Directory, Rename, Restore Archive, Run, Selective Archive, Set Protection, Update Request Set, and Volume Archive.

Tips

The following paragraphs present some tips for creating installation scripts.

1. Do not path anywhere within the installation script. Since you can't know what the user's original current working directory was, you can't path back to it. You will surprise the user by leaving him or her in a new path.

2. Use message files rather than putting messages directly into the installation script. This method aids nationalization. In addition, if verbose mode is used, messages wrap around the screen and scroll up, so the messages are hard to read.

3. Take care not to prompt for information in your script that the Installation Manager has already prompted for (the name of the command file or the Context Manager configuration file, for example).

4. If you give the user the option to terminate during the script, use the Batch Cancel statement. When an installation terminates abnormally, Batch needs to notify the Installation Manager not to update the database. If you do not use the Batch command Cancel statement, Batch will have no way of knowing that the script was terminated abnormally and the database could be incorrectly updated. When Batch encounters a Cancel statement, it causes the Installation Manager to report status code 4 (operator intervention).

5. If your product has to be installable by version 12.0 of the Installation Manager, you cannot use subpackages. To make your installation script compatible with both versions, your installation

script must perform the extra work of assigning values to those installation variables that are not available in version 12.0. Use the Batch FileVersion statement to determine what version of the Installation Manager run file you have. The variables that you have to set are DirectoryTo, MsgFile, and VolumeTo.

FileVersion returns a string. It should be checked as follows:

```
ver = FileVersion<'[Sys]<Sys>InstallMgr.run')
If Substring('12.0',ver)<>65535
  ...
EndIf
```

By calling Substring instead of checking to see if ver = ’12.0’ you are assured of obtaining the correct answer, regardless of the version of 12.0 that the user may have installed (Substring would report TRUE when the version is 12.0.1, where "if ver = ’12.0’ would not.)

6. Installation scripts that are accessed via the $Call command and that have RestartLabel directives in them must be copied somewhere onto a hard disk. Batch terminates with status code 302 (write-protected) if the script file is referenced on your write-protected distribution diskette. When Batch encounters a RestartLabel statement, it attempts to open the script file in mode modify, which it cannot do if the media is write-protected. The Installation Manager copies your main installation script to hard disk, so you don't have to worry about copying it.

7. If, during testing, your script fails with "Command not found," then you are using a command that is not contained in *Install>English.cmds*. In that case your script must add the command using the New Command command before it issues the $Command statement for that command.

Installation Script Examples

The following pages illustrate three sets of installation files. Each JCL installation script file begins on a lefthand page. Numbered comments accompanying each script correspond to script lines. The facing righthand page includes comments for the numbered lines in the script. These examples are not intended to teach you how to write JCL, but instead, how to use the installation variables to best advantage.

The binary message file is created from a text file. The contents of the text file is shown, even though the file is labeled **InstallMsg.bin.*

The values of the installation variables are given as if the user had not changed them from their default values. For example, VolumeTo will be described as having the value [Sys] for private installations. As stated above, you as script writer cannot count on any specific value being in the installation variables, but instead must use them to carry out the desires of the user.

Example 1: One Subpackage

This example contains one subpackage and spans two floppy disks. The installation files are *Install.ctrl, InstallMsg.bin, Install.cmds,* and *Install.jcl.* They are placed in the <Sys> directory of the first disk. A zero-length file, 'Tape Utilities Diskette 1 of 2' is put in the <Sys> directory on the first disk. A zero-length file, 'Tape Utilities Diskette 2 of 2' is put in the <Sys> directory on the second disk. Product files are placed in the <CTOS> directory of the two disks.

Install.ctrl File

```
:PackageName:Basic Tape
:Version:12.1.0
:RequiredDiskSpace:635
:Verify:
:Directory:<Sys>
:Commands:'Configure Sequential Access Device' 'DeInstall
Sequential Access Service' 'Install Sequential Access
Service' 'Tape Copy' 'Tape Erase' 'TapeRetension'
:ProductFiles:InstallSeqService.run NGenSeqService.run
QicConfig.sys SeqAccessCopy.run SeqAccessUtility.run
SeqServiceMsg.bin
```

(*QicConfig.sys* is not placed in *OverwriteNoFiles* because in this release, the format of the file has changed. Files in the old format will no longer work, and therefore must be overwritten.)

InstallMsg.bin File

```
:1: "Tape Utilities Diskette 2 of 2"
:2: "Merging commands... "
:3: "Copying files... "
:4: "done."
```

Install.cmds File

The Command File Editor was used to create the six commands listed in the control file.

Install.jcl File

Number **Script**

```
        ; Install.jcl
        ; Install the Basic Tape subpackage

        ; Except for yes/no strings, this message is the
        ; only nationalizable string in this file.

1       ErrorMsg  = 'Cannot access message file'

2       FromFloppy   = 0
        FromServer   = 2
3       FloppySpec   = ConcatStrings(DeviceFrom,'<CTOS>')
        FromSpec     = ConcatStrings(FloppySpec, '*')
        ToSpec       = ConcatStrings(VolumeTo, DirectoryTo)

4       InitErc = InitMsgFile(MsgFile)
        If InitErc > 0
            DisplayLine(ErrorMsg)
            Cancel
        Endif

5       CopyMsg = GetMsg(3)
        DoneMsg = GetMsg(4)
        Msg = GetMsg(2)
        Display(Msg)

6       Command Merge Command Files  ,&
            CmdFileFrom              ,&
            CmdFileTo                ,&
            Yes                      ,&
            VolumeTo

        DisplayLine(DoneMsg)
        DisplayLine
        Display(CopyMsg)
```

Number **Comment**

When values for installation variables are given, it is assumed that the user has not used the Examine/Change Defaults option from the Installation Defaults menu. For floppy disk installations, it is assumed that the user has inserted the floppy disk in drive 0.

1. Assign strings to be used later.

2. Assign values to enhance readability.

3. DeviceFrom is [f0] if the installation is private, [!Sys] if the installation is public. VolumeTo is [Sys] if the installation is private, [!Sys] if public. DirectoryTo is <Sys>.

4. Initialize the message file. The message file is *[Scr]<$>InstallMsg.bin* if the installation is private or *[!Sys]<Installed>BasicTape>12.1.0>InstallMsg.bin* if the installation is from the server. Print a message if the file does not exist. It should always be there–this is just defensive coding. The Cancel statement causes termination of script processing by Batch. The installation will fail because the product files were not updated.

5. Since the "copying" and "done" messages are used more than once, put them in their own variables.

6. Merge the command files. CmdFileFrom is *[Scr]<$>Install.cmds* if the installation is private or *[!Sys]<Installed>BasicTape>12.1.0>Install.cmds* if the installation is from the server. CmdFileTo is *[Sys]<Sys>Sys.cmds* if the installation is private, *[!Sys]<Sys>Cluster.cmds* if it is public. VolumeTo is [Sys] if the installation is private, [!Sys] if it is public.

Number	Script

```
7     If InstallType = FromServer
          Command LCopy                       ,&
                                              ,&
              @[Scr]<$>InstallOverwriteOkFrom.fls,&
              @[Scr]<$>InstallOverwriteOkTo.fls,&
                                              ,&
              Yes                             ,&
              No                              ,&
              No                              ,&
              Yes

      DisplayLine(DoneMsg)
      DisplayLine

8         GoTo Endit
      Endif

      RestartLabel

9     Command LCopy                           ,&
          FromSpec                            ,&
                                              ,&
          ToSpec                              ,&
                                              ,&
              Yes                             ,&
              No                              ,&
              No                              ,&
              Yes

      DisplayLine(DoneMsg)
      DisplayLine

      RestartLabel
```

Number **Comment**

7. When the user does a public installation, this script file, along with the accompanying command and message files will be copied to [!Sys]<Installed> with the prefix BasicTape>12.1.0>. The script will be invoked when the user attempts to install the product from the server. Therefore it (and indeed every script) must allow for the case when the user installs from the server (even if the script displays a message saying that installation from the server is not supported for the subpackage). This **LCopy** command is performed if the user invoked the Installation Manager using the Executive command **Server Install** or if the "Install from server" option was chosen from the Installation Media menu. *[Scr]<$>InstallOverwriteOkFrom.fls* will contain:

   [!Sys]<Sys>InstallSeqService.run
   [!Sys]<Sys>NGenSeqService.run
   [!Sys]<Sys>QicConfig.sys
   [!Sys]<Sys>SeqAccessCopy.run
   [!Sys]<Sys>SeqAccessUtility.run
   [!Sys]<Sys>SeqServiceMsg.bin

   [Scr]<$>InstallOverwriteOkTo.fls will contain:

   [Sys]<Sys>InstallSeqService.run
   [Sys]<Sys>NGenSeqService.run
   [Sys]<Sys>QicConfig.sys
   [Sys]<Sys>SeqAccessCopy.run
   [Sys]<Sys>SeqAccessUtility.run
   [Sys]<Sys>SeqServiceMsg.bin

8. Since there is nothing more to do in the case of an installation from the server, quit. A GoTo statement is used to make the script more readable. An Else clause would have been equally correct.

9. Everything from this point gets executed only if the installation is being done from floppy disks. Copy the files from the first disk. FromSpec is [f0]<CTOS>* and ToSpec is [Sys]<Sys> if the installation is private or [!Sys]<Sys> if the installation is public.

Number	Script

```
10     Msg = GetMsg(1)
       NextFloppy(Msg)

       DisplayLine
       Display(CopyMsg)

11     Command LCopy                        ,&
            FromSpec                        ,&
                                            ,&
            ToSpec                          ,&
                                            ,&
                 Yes                        ,&
                 No                         ,&
                 No                         ,&
                 Yes

       DisplayLine(DoneMsg)
       DisplayLine

       Endit:
       End
```

Number	Comment
10.	Ask the user to insert the next floppy disk.
11.	Copy the files from the second disk. FromSpec and ToSpec remain the same.

Example 2: Nested Subpackages

This second example contains two levels of subpackages and spans three floppy disks. The installation files are

Install.ctrl
DevelopmentUtilities>Install.ctrl

DevelopmentRunFiles>Install.ctrl
DevelopmentRunFiles>Install.jcl
DevelopmentRunFiles>InstallMsg.bin
DevelopmentRunFiles>Install.cmds

DevelopmentLibraries>Install.ctrl
DevelopmentLibraries>Install.jcl
DevelopmentLibraries>InstallMsg.bin

AsynchronousExamples>Install.ctrl
AsynchronousExamples>Install.jcl
AsynchronousExamples>InstallMsg.bin
AsynchronousExamples>Install.cmds

These files are placed in the <Sys> directory of the first disk.

The following files are also placed in the <Sys> directory of the second disk:

DevelopmentLibraries>Install.ctrl
DevelopmentLibraries>Install.jcl
DevelopmentLibraries>InstallMsg.bin

The following files are also placed in the <Sys> directory of the third disk:

AsynchronousExamples>Install.ctrl
AsynchronousExamples>Install.jcl
AsynchronousExamples>InstallMsg.bin
AsynchronousExamples>Install.cmds

A zero-length file, 'Development Package Diskette 1 of 3' is put in the <Sys> directory on the first disk. A zero-length file, 'Development Package Diskette 2 of 3' is put in the <Sys> directory on the second disk.

A zero-length file, 'Development Package Diskette 3 of 3' is put in the <Sys> directory on the third disk.

Product files for Development Run Files are placed in the <CTOS> directory of disks 1 and 2. Product files for Development Libraries are placed in the <Lib> directory of disks 2 and 3. Product files for Asynchronous examples are placed in the <Async> directory of disk 3.

Install.ctrl File

```
:PackageName:Development Package
:MultiPkgName:Development Utilities
:MultiPkgName:Asynchronous Examples
```

DevelopmentUtilities>Install.ctrl File

```
:PackageName:Development Utilities
:MultiPkgName:Development Run Files
:MultiPkgName:Development Libraries
```

Note that Install.ctrl could have been designed to look like:

```
:PackageName:Development Package
:MultiPkgName:Development Run Files
:MultiPkgName:Development Libraries
:MultiPkgName:Asynchronous Examples
```

The first way groups the Run Files and Libraries into their own menu, which is a nicer presentation to the user. The second way requires DevelopmentUtilities>Install.ctrl to be removed. Both ways would result in the same entries in the installation database.

Note that the Installation Manager installs subpackages in the order given in the various control files. If all three subpackages are chosen, they would be installed in the following order: (1) Development Run Files; (2) Development Libraries; (3) Asynchronous Examples.

In this example, *Install.ctrl* would be opened, and the menu would contain two choices: Development Utilities and Asynchronous Examples. If the user chooses both, *DevelopmentUtilities>Install.ctrl* will be opened and the menu would contain the two choices Development Run Files and Development Libraries. If the user chooses both, *DevelopmentRun-Files>Install.ctrl* will be opened. Since this is the bottom of the "tree," the associated command, script, and message files would be copied to [Scr]<$> and the installation would be performed. Then *DevelopmentLibraries>Install.ctrl* would be opened. Again this is the bottom of the tree, so the files would be copied and the installation performed. Then *AsynchronousExamples>Install.ctrl* would be opened, the files copied and that installation performed.

If *DevelopmentRunFiles>Install.ctrl*, *DevelopmentLibraries>Install.ctrl*, or *AsynchronousExamples>Install.ctrl* had not been at the bottom of the tree (in other words, they contained :MultiPkgName: entries), then menus would have been presented to the user at the time the files were opened. The subpackages listed in *DevelopmentRunFiles>Install.ctrl* would be installed before those listed in *DevelopmentLibraries>Install.ctrl*, and so on. This can go on for a theoretically infinite number of layers.

DevelopmentRunFiles>Install.ctrl **File**

```
:PackageName:Development Run Files
:Version:12.1.0
:Verify:
:RequiredDiskSpace:1400
:Directory:<Sys>
:Commands:'Assemble' 'Bind' 'Dump'
:ProductFiles:Assembler.run Linker.run Dump.run
```

DevelopmentRunFiles>InstallMsg.bin **File**

```
:1: "Development Package diskette 2 of 3"
:2: "Development Package diskette 3 of 3"
:3: "Development Libraries"
:4: "Asynchronous Examples"
:5: "Merging commands... "
:6: "Copying files... "
:7: "done."
```

DevelopmentRunFiles>Install.cmds **File**

The Command File Editor is used to create the three commands listed in the control file.

DevelopmentRunFiles>Install.jcl File

Number **Script**

```
      ; DevelopmentRunFiles>Install.jcl
      ; Install the Development Run Files subpackage

      ; Except for yes/no strings, this message is the
      ; only nationalizable string in this file.

1     ErrorMsg     = 'Cannot access message file'

2     FromFloppy   = 0
      FromServer   = 2
3     FloppySpec   = ConcatStrings(DeviceFrom,'<CTOS>')
      FromSpec     = ConcatStrings(FloppySpec, '*')
      ToSpec       = ConcatStrings(VolumeTo, DirectoryTo)

4     InitErc = InitMsgFile(MsgFile)
      If InitErc > 0
          DisplayLine(ErrorMsg)
          Cancel
      Endif

5     CopyMsg = GetMsg(6)
      DoneMsg = GetMsg(7)
      Msg = GetMsg(5)
      Display(Msg)

6     Command Merge Command Files  ,&
          CmdFileFrom              ,&
          CmdFileTo                ,&
          Yes                      ,&
          VolumeTo

      Displayline(DoneMsg)
      DisplayLine
      Display(CopyMsg)
```

Number	Comment
	When values for installation variables are given, it is assumed that the user has not used the Examine/Change Defaults option from the Installation Defaults menu. For floppy disk installations, it is assumed that the user has inserted the floppy disk in drive 0.
1.	Assign strings to be used later.
2.	Assign values to enhance readability.
3.	DeviceFrom is [f0] if the installation is private, [!Sys] if the installation is public. VolumeTo is [Sys] if the installation is private, [!Sys] if public. DirectoryTo is <Sys>.
4.	Initialize the message file. The message file is *[Scr]<$>DevelopmentRunFiles>InstallMsg.bin* if the installation is private or *[!Sys]<Installed>DevelopmentRunFiles>12.1.0>InstallMsg.bin* if the installation is from the server. Print a message if the file does not exist. It should always be there–this is just defensive coding. The Cancel statement causes termination of script processing by Batch. The installation will fail because the product files were not updated.
5.	Since the "copying" and "done" messages are used more than once, put them in their own variables.
6.	Merge the command files. CmdFileFrom is *[Scr]<$>DevelopmentRunFiles>Install.cmds* if the installation is private or *[!Sys]<Installed>DevelopmentRunFiles>12.1.0>Install.cmds* if the installation is from the server. CmdFileTo is *[Sys]<Sys>Sys.cmds* if the installation is private, *[!Sys]<Sys>Cluster.cmds* if it is public. VolumeTo is [Sys] if the installation is private, [!Sys] if it is public.

Number	Script

```
 7     If InstallType = FromServer
           Command LCopy                ,&
                                        ,&
               @[Scr]<$>InstallOverwriteOkFrom.fls,&
               @[Scr]<$>InstallOverwriteOkTo.fls,&
                                        ,&
               Yes                      ,&
               No                       ,&
               No                       ,&
               Yes

       Displayline(DoneMsg)
       DisplayLine

 8         GoTo Endit
       Endif

       RestartLabel

 9     Command LCopy                    ,&
           FromSpec                     ,&
                                        ,&
           ToSpec                       ,&
                                        ,&
               Yes                      ,&
               No                       ,&
               No                       ,&
               Yes

       Displayline(DoneMsg)
       DisplayLine

       RestartLabel

10     Msg = GetMsg(1)
       NextFloppy(Msg)

       DisplayLine
       Display(CopyMsg)
```

Number **Comment**

7. When the user does a public installation, this script file, along with the accompanying command and message files will be copied to [!Sys]<Installed> with the prefix DevelopmentRunFiles>12.1.0>. The script will be invoked when the user attempts to install the product from the server. Therefore it (and indeed every script) must allow for the case when the user installs from the server (even if the script displays a message saying that installation from the server is not supported for the subpackage). This **LCopy** command is performed if the user invoked the Installation Manager using the Executive command **Server Install** or if the "Install from server" option was chosen from the Installation Media menu.

   [Scr]<$>InstallOverwriteOkFrom.fls will contain:

   [!Sys]<Sys>Assembler.run
   [!Sys]<Sys>Linker.run
   [!Sys]<Sys>Dump.run

   [Scr]<$>InstallOverwriteOkTo.fls will contain:

   [Sys]<Sys>Assembler.run
   [Sys]<Sys>Linker.run
   [Sys]<Sys>Dump.run

8. Since there is nothing more to do in the case of an installation from the server, quit. A GoTo statement is used to make the script more readable. An Else clause would have been equally correct.

9. Everything from this point is executed only if the installation is being done from floppy disks. Copy the files from the first disk. FromSpec is [f0]<CTOS>* and ToSpec is [Sys]<Sys> if the installation is private or [!Sys]<Sys> if the installation is public.

10. Ask the user to insert the next floppy disk.

Number **Script**

```
11    Command LCopy                         ,&
          FromSpec                          ,&
                                            ,&
          ToSpec                            ,&
                                            ,&
              Yes                           ,&
              No                            ,&
              No                            ,&
              Yes

      Displayline(DoneMsg)
      DisplayLine

12    Pkg = GetMsg(3)
      If SubString(Pkg,Pkgs) <> 65535
          Msg = GetMsg(1)
          NextFloppy(Msg)
          GoTo Endit
      Endif

13    Pkg = GetMsg(4)
      If SubString(Pkg,Pkgs) <> 65535
          Msg = GetMsg(2)
          NextFloppy(Msg)
      Endif

      Endit:
      End
```

Number	Comment

11. Copy the files from the second disk. FromSpec and ToSpec remain the same.

12. See if the user has chosen the Development Libraries subpackage. If it was chosen, the NextFloppy will not ask the user to insert a floppy disk, since it is already inserted. The statement is here so in case the subpackage grows in the future, minimal rewriting will have to be done to the script. The GoTo is required in case the user also chose the Asynchronous Examples subpackage. (If the GoTo were missing and the Asynchronous Examples subpackage were chosen, the user would be prompted to insert the third disk. This would cause the Installation Manager to ask the user to mount the first disk of the Development Libraries subpackage, because it would be looking for *[f0]<Sys>DevelopmentLibraries>Install.ctrl*, which is on the second disk.)

13. In the case that the user did not choose the Development Libraries subpackage, ascertain whether the Asynchronous Examples subpackage was chosen. If so, ask the user to insert the third disk.

DevelopmentLibraries>Install.ctrl File

```
:PackageName:Development Libraries
:Version:12.1.0
:Verify:
:RequiredDiskSpace:800
:Directory:<Sys>
:ProductFiles:Ctos.lib Enls.lib Async.lib
```

DevelopmentLibraries>InstallMsg.bin File

```
:1: "Development Package diskette 3 of 3"
:2: "Copying files... "
:3: "done."
```

DevelopmentLibraries>Install.jcl **File**

Number **Script**

```
      ; DevelopmentLibraries>Install.jcl
      ; Install the Development Libraries subpackage

      ; Except for yes/no strings, these two messages are
      ; the only nationalizable strings in this file.

1     ErrorMsg      ='Cannot access message file'
      FirstDisk     ='Development Package diskette 2 of 3'

2     FromFloppy    = 0
      FromServer    = 2
3     FloppySpec    = ConcatStrings(DeviceFrom, '<Lib>')
      FromSpec      = ConcatStrings(FloppySpec, '*')
      ToSpec        = ConcatStrings(VolumeTo, DirectoryTo)

4     If InstallType = FromFloppy
          NextFloppy(FirstDisk)
      Endif

5     InitErc = InitMsgFile(MsgFile)
      If InitErc > 0
          DisplayLine(ErrorMsg)
          Cancel
      Endif

6     CopyMsg = GetMsg(2)
      DoneMsg = GetMsg(3)
      Display(CopyMsg)
```

Number **Comment**

When values for installation variables are given, it is assumed that the user has not used the Examine/Change Defaults option from the Installation Defaults menu. For floppy disk installations, it is assumed that the user has inserted the floppy disk in drive 0.

1. Assign strings to be used later.

2. Assign values to enhance readability.

3. DeviceFrom is [f0] if the installation is private, [!Sys] if the installation is public. VolumeTo is [Sys] if the installation is private, [!Sys] if public. DirectoryTo is <Sys>.

4. Make sure that the correct floppy disk is inserted. If the user did not choose "Development Run Files," then the first disk is still inserted.

5. Initialize the message file. The message file is *[Scr]<$>DevelopmentLibraries>InstallMsg.bin* if the installation is private or *[!Sys]<Installed>DevelopmentLibraries>12.1.0>InstallMsg.bin* if the installation is from the server. Print a message if the file does not exist. It should always be there–this is just defensive coding. The Cancel statement causes termination of script processing by Batch. The installation will fail because the product files were not updated.

6. Since the "copying" and "done" messages are used more than once, put them in their own variables.

Number **Script**

```
 7     If InstallType = FromServer
           Command LCopy                    ,&
                                            ,&
               @[Scr]<$>InstallOverwriteOkFrom.fls,&
               @[Scr]<$>InstallOverwriteOkTo.fls,&
                                            ,&
               Yes                          ,&
               No                           ,&
               No                           ,&
               Yes

       Displayline(DoneMsg)
       DisplayLine

 8         GoTo Endit
       Endif

       RestartLabel

 9     Command LCopy                        ,&
           FromSpec                         ,&
                                            ,&
           ToSpec                           ,&
                                            ,&
               Yes                          ,&
               No                           ,&
               No                           ,&
               Yes

       Displayline(DoneMsg)
       DisplayLine

       RestartLabel

10     Msg = GetMsg(1)
       NextFloppy(Msg)

       DisplayLine
       Display(CopyMsg)
```

Number **Comment**

7. When the user does a public installation, this script file, along with the accompanying command and message files will be copied to [!Sys]<Installed> with the prefix DevelopmentLibraries>12.1.0>. The script will be invoked when the user attempts to install the product from the server. Therefore it (and indeed every script) must allow for the case when the user installs from the server (even if the script displays a message saying that installation from the server is not supported for the subpackage). This **LCopy** command is performed if the user invoked the Installation Manager using the Executive command **Server Install** or if the "Install from server" option was chosen from the Installation Media menu.

   [Scr]<$>InstallOverwriteOkFrom.fls will contain:

   [!Sys]<Sys>Ctos.lib
   [!Sys]<Sys>Enls.lib
   [!Sys]<Sys>Async.lib

   [Scr]<$>InstallOverwriteOkTo.fls will contain:

   [Sys]<Sys>Ctos.lib
   [Sys]<Sys>Enls.lib
   [Sys]<Sys>Async.lib

8. Since there is nothing more to do in the case of an installation from the server, quit. A GoTo statement is used to make the script more readable. An Else clause would have been equally correct.

9. Everything from this point gets executed only if the installation is being done from floppy disks. Copy the files from the first disk. FromSpec is [f0]<Lib>* and ToSpec is [Sys]<Sys> if the installation is private or [!Sys]<Sys> if the installation is public.

10. Ask the user to insert the next floppy disk.

Number	Script

```
11    Command LCopy                          ,&
          FromSpec                           ,&
                                             ,&
          ToSpec                             ,&
                                             ,&
              Yes                            ,&
              No                             ,&
              No                             ,&
              Yes

      Endit:
      End
```

Number	Comment
11.	Copy the files from the third disk. FromSpec and ToSpec remain the same. There is no need to see if the user has chosen the Asynchronous Examples subpackage, since the third disk is already in the disk drive.

AsynchronousExamples>Install.ctrl File

```
:PackageName:Asynchronous Examples
:Version:12.1.0
:Verify:
:RequiredDiskSpace:360
:Directory:<Async>
:Commands:Start Stop Example
:ProductFiles:Async.h AsyncService.c Deinstall.c
Example.c ExDef.h ExFunc.h ExRqblk.h ExRqLabl.asm
LinkDeinstall.fls LinkDeinstall.sub LinkExample.fls
LinkExample.sub LinkStart.fls LinkStart.sub LinkStop.fls
LinkStop.sub ReadMe RequestEx.txt Start.c Stop.c
```

AsynchronousExamples>InstallMsg.bin **File**

```
:1: "Listing requests... "
:2: "Making Request.sys... "
:3: "Merging commands... "
:4: "Copying files... "
:5: "done."
```

AsynchronousExamples>Install.cmds **File**

The Command File Editor was used to create the three commands listed in the control file.

AsynchronousExamples>Install.jcl File

Number **Script**

```
      ; AsynchronousExamples>Install.jcl
      ; Install the Asynchronous Examples subpackage

      ; Except for yes/no strings, these two messages are
      ; the only nationalizable strings in this file.

1     ErrorMsg     = 'Cannot access message file'
      FirstDisk    ='Development Package diskette 3 of 3'

2     FromFloppy = 0
      FromServer = 2

      TotalSilence = 2


3     FloppySpec = ConcatStrings(DeviceFrom, '<Async>')
      FromSpec   = ConcatStrings(FloppySpec, '*')
      ToSpec     = ConcatStrings(VolumeTo, DirectoryTo)
      ScrSpec    = '[Scr]<$>'
      If Public
         SysSpec = '[!Sys]<Sys>'
      Else
          SysSpec= '[Sys]<Sys>'
      Endif
      RqSpec     = ConcatStrings(SysSpec,'Request.sys')
      RqListSpec = ConcatStrings(ScrSpec,'LRS')
      RqTextSpec = ConcatStrings(ToSpec,'Request.Ex.txt')

4     If InstallType = FromFloppy
          NextFloppy(FirstDisk)
      Endif

5     InitErc = InitMsgFile(MsgFile)
      If InitErc > 0
          DisplayLine(ErrorMsg)
          Cancel
      Endif
```

```
6    DoneMsg = GetMsg(5)
     Msg = GetMsg(3)
     Display(Msg)
```

Number	Comment
	When values for installation variables are given, it is assumed that the user has not used the Examine/Change Defaults option from the Installation Defaults menu. For floppy disk installations, it is assumed that the user has inserted the floppy disk in drive 0.
1.	Assign strings to be used later.
2.	Assign values to enhance readability.
3.	DeviceFrom is [f0] if the installation is private, [!Sys] if the installation is public. VolumeTo is [Sys] if the installation is private, [!Sys] if public. DirectoryTo is <Sys>. Note that instead of using VolumeTo as the volume part of SysSpec, we explicitly set it to [Sys]<Sys> or [!Sys]<Sys>. The user has the option of changing the destination volume to something other than [Sys] or [!Sys], but the request file is always on the Sys volume of the target computer.
4.	Make sure that the correct floppy disk is inserted. If the user chose only Asynchronous Examples, then the first disk is still inserted.
5.	Initialize the message file. The message file is *[Scr]<$>AsynchronousExamples>InstallMsg.bin* if the installation is private or *[!Sys]<Installed>AsynchronousExamples>12.1.0>InstallMsg.bin* if the installation is from the server. Print a message if the file does not exist. It should always be there–this is just defensive coding. The Cancel statement causes termination of script processing by Batch. The installation will fail because the product files were not updated.
6.	Since the "copying" and "done" messages are used more than once, put them in their own variables.

Number	Script

7

```
Command Merge Command Files  ,&
    CmdFileFrom              ,&
    CmdFileTo                ,&
    Yes                      ,&
    VolumeTo

Displayline(DoneMsg)
DisplayLine
Msg = GetMsg(4)
Display(Msg)
```

8

```
If InstallType = FromServer
    Command LCopy            ,&
                             ,&
        @[Scr]<$>InstallOverwriteOkFrom.fls,&
        @[Scr]<$>InstallOverwriteOkTo.fls,&
                             ,&
        Yes                  ,&
        No                   ,&
        No                   ,&
        Yes
```

Number **Comment**

7. Merge the command files. CmdFileFrom is *[Scr]<$>AsynchronousExamples>Install.cmds* if the installation is private or *[!Sys]<Installed>AsynchronousExamples>12.1.0>Install.cmds* if the installation is from the server. CmdFileTo is *[Sys]<Sys>Sys.cmds* if the installation is private, *[!Sys]<Sys>Cluster.cmds* if it is public. VolumeTo is [Sys] if the installation is private, [!Sys] if it is public.

8. When the user does a public installation, this script file, along with the accompanying command and message files will be copied to [!Sys]<Installed> with the prefix AsynchronousExamples>12.1.0>. The script will be invoked when the user attempts to install the product from the server. Therefore it (and indeed every script) must allow for the case when the user installs from the server (even if the script displays a message saying that installation from the server is not supported for the subpackage). This **LCopy** command is performed if the user invoked the Installation Manager using the Executive command **Server Install** or if the "Install from server" option was chosen from the Installation Media menu.

   [Scr]<$>InstallOverwriteOkFrom.fls will contain:

   [!Sys]<Async>Async.h
   [!Sys]<Async>AsyncService.c
   [!Sys]<Async>Deinstall.c
   [!Sys]<Async>Example.c
   [!Sys]<Async>ExDef.h
   [!Sys]<Async>ExFunc.h
   [!Sys]<Async>ExRqblk.h
   [!Sys]<Async>ExRqLabl.asm
   [!Sys]<Async>LinkDeinstall.fls
   [!Sys]<Async>LinkDeinstall.sub
   [!Sys]<Async>LinkExample.fls
   [!Sys]<Async>LinkExample.sub
   [!Sys]<Async>LinkStart.fls
   [!Sys]<Async>LinkStart.sub
   [!Sys]<Async>LinkStop.fls

Number	Script

```
 9          GoTo DoRqs
      Endif

      RestartLabel

10    Command LCopy                        ,&
          FromSpec                         ,&
                                           ,&
          ToSpec                           ,&
                                           ,&
              Yes                          ,&
              No                           ,&
              No                           ,&
              Yes

      DoRqs:

      Displayline(DoneMsg)
      DisplayLine

      RestartLabel

      Msg = GetMsg(1)
      Display(Msg)
```

Number **Comment**

[!Sys]<Async>LinkStop.sub
[!Sys]<Async>ReadMe
[!Sys]<Async>RequestEx.txt
[!Sys]<Async>Start.c
[!Sys]<Async>Stop.c

[Scr]<$>InstallOverwriteOkTo.fls will contain:

[Sys]<Async>Async.h
[Sys]<Async>AsyncService.c
[Sys]<Async>Deinstall.c
[Sys]<Async>Example.c
[Sys]<Async>ExDef.h
[Sys]<Async>ExFunc.h
[Sys]<Async>ExRqblk.h
[Sys]<Async>ExRqLabl.asm
[Sys]<Async>LinkDeinstall.fls
[Sys]<Async>LinkDeinstall.sub
[Sys]<Async>LinkExample.fls
[Sys]<Async>LinkExample.sub
[Sys]<Async>LinkStart.fls
[Sys]<Async>LinkStart.sub
[Sys]<Async>LinkStop.fls
[Sys]<Async>LinkStop.sub
[Sys]<Async>ReadMe
[Sys]<Async>RequestEx.txt
[Sys]<Async>Start.c
[Sys]<Async>Stop.c

9. The next step when installing from the server is merging the requests.

10. Copy the files from the disk. FromSpec is [f0]<Async>* and ToSpec is [Sys]<Async> if the installation is private or [!Sys]<Async> if the installation is public.

Number **Script**

```
 11    Command List Request Set      ,&
           RqSpec                    ,&
           RqListSpec

       Displayline(DoneMsg)
       DisplayLine
       Msg = GetMsg(2)
       Display(Msg)

 12    SaveVidLevel = VideoLevel
       If VideoLevel = TotalSilence
           EchoSome
       EndIf

Command Make Request Set,&
           (RqListSpec, RqTextSpec) ,&
           RqSpec                    ,&
                                     ,&
           12.1.0

If SaveVidLevel = TotalSilence
       EchoOff
EndIf

       Displayline(DoneMsg)
       DisplayLine

       End
```

Number	Comment
11.	List the request set. RqSpec is *[Sys]<Sys>Request.sys* if the installation is private or *[!Sys]<Sys>Request.sys*. RqListSpec is [Scr]<$>LRS.
12.	Merge the new requests into the user's request file. Note that video must be turned on, since there might be errors reported by Make Request Set. If it reports errors, it stops for user input. If video were off (non-verbose), the installation would appear to hang, since the user would never see the messages. RqListSpec and RqSpec are unchanged. RqTextSpec is *[Sys]<Async>Request.Ex.txt* if the installation is private or *[!Sys]<Async>Request.Ex.txt* if the installation is public.

Example 3: Tape Installation

This last example is a tape installation of the second example (Development Package). In this example, the following events occur:

1. The product files are copied for each of the chosen subpackages.

2. Two of the subpackages have associated command files (Development Run Files and Asynchronous Examples). If one (or both) of these subpackages is chosen, its command files are merged.

3. One of the subpackages (Asynchronous Examples) has commands that need to be merged if that subpackage is chosen.

The string variable *pkgs* contains the names of all the subpackages chosen by the user. Install Manager passes this variable to Batch. The JCL installation script then calls the Batch function $Substring to query this variable.

The installation files are

Install.ctrl (same as *DevelopmentUtilities>Install.ctrl* in the previous example)
Install.jcl (see following pages)
InstallMsg.bin (see following pages)

DevelopmentRunFiles>Install.ctrl
DevelopmentRunFiles>Install.cmds

DevelopmentLibraries>Install.ctrl

AsynchronousExamples>Install.ctrl
AsynchronousExamples>Install.cmds

These files are placed in [Qic]0.

The contents of the other installation files are the same as that of the files in the previous example.

InstallMsg.bin File

```
:1: "Listing requests... "
:2: "Making Request.sys... "
:3: "Merging commands... "
:4: "Restoring files... "
:5: "done."
:6: "Asynchronous Examples"
:7: "AsynchronousExamples>Install.cmds"
:8: "Development Run Files"
:9: "DevelopmentRunFiles>Install.cmds"
```

Install.jcl File

Number **Script**

```
      ; Install.jcl
      ; Install Development Package from Qic Tape

      ; Except for yes/no strings, this message is the
      ; only nationalizable string in this file.

 1    ErrorMsg     = 'Cannot access message file'

 2   ScrSpec     = '[Scr]<$>'
     ToSpec      = ConcatStrings(VolumeTo, DirectoryTo)
     If Public
        SysSpec = '[!Sys]<Sys>'
     Else
        SysSpec = '[Sys]<Sys>'
     Endif
     RqSpec      = ConcatStrings(SysSpec,'Request.sys')
     RqListSpec = ConcatStrings(ScrSpec,  'LRS')
     RqTextSpec =ConcatStrings(ToSpec,'Request.vm.txt')

 3    InitErc = InitMsgFile(MsgFile)
      If InitErc > 0
          DisplayLine(ErrorMsg)
          Cancel
      Endif

 4    DoneMsg = GetMsg(5)
      Msg = GetMsg(4)
      Display(Msg)

      RestartLabel
```

Number	Comment
	When values for installation variables are given, it is assumed that the user has not used the Examine/Change Defaults option from the Installation Defaults menu. Note that in this script, we don't have to worry about floppy and server installations, since the script file will only be used for a tape installation.
1.	Assign strings to be used later.
2.	VolumeTo is [Sys] if the installation is private, [!Sys] if public. DirectoryTo is <Sys>. Note that instead of using VolumeTo as the volume part of SysSpec, we explicitly set it to [Sys]<Sys> or [!Sys]<Sys>. The user has the option of changing the destination volume to something other than [Sys] or [!Sys], but the request file is always always on the sys volume of the target computer.
3.	Initialize the message file. The message file is *[Scr]<$>InstallMsg.bin*. Print a message if the file does not exist. It should always be there–this is just defensive coding. The Cancel statement causes termination of script processing by Batch. The installation will fail because the product files were not updated.
4.	Since the "copying" and "done" messages are used more than once, put them in their own variables.

Number	Script

```
5    Command Restore Archive      ,&
                DeviceFrom         ,&
                @[Scr]<$>InstallOverwriteOkFrom.fls,&
                @[Scr]<$>InstallOverwriteOkTo.fls,&
                yes

     Displayline(DoneMsg)
     DisplayLine

     RestartLabel
```

Number **Comment**

5. Restore the product files. We will assume that the user chose all three subpackages. Since **Restore Archive** will create directories that do not already exist, we do not have to create [VolumeTo]<Async>. DeviceFrom is [Qic]1.

[Scr]<$>InstallOverwriteOkFrom.fls will contain:

<>Assembler.run*
<>Linker.run*
<>Dump.run*
<>Ctos.lib*
<>Enls.lib*
<>Async.lib*
<Async>Async.h
<>AsyncService.c*
<>Deinstall.c*
<>Example.c*
<>ExDef.h*
<>ExFunc.h*
<>ExRqblk.h*
<>ExRqLabl.asm*
<>LinkDeinstall.fls*
<>LinkDeinstall.sub*
<>LinkExample.fls*
<>LinkExample.sub*
<>LinkStart.fls*
<>LinkStart.sub*
<>LinkStop.fls*
<>LinkStop.sub*
<>ReadMe*
<>RequestEx.txt*
<>Start.c*
<>Stop.c*

[Scr]<$>InstallOverwriteOkTo.fls will contain:

[Sys]<Sys>Assembler.run
[Sys]<Sys>Linker.run
[Sys]<Sys>Dump.run
[Sys]<Sys>Ctos.lib
[Sys]<Sys>Enls.lib
[Sys]<Sys>Async.lib
[Sys]<Async>Async.h
[Sys]<Async>AsyncService.c
[Sys]<Async>Deinstall.c
[Sys]<Async>Example.c
[Sys]<Async>ExDef.h
[Sys]<Async>ExFunc.h
[Sys]<Async>ExRqblk.h
[Sys]<Async>ExRqLabl.asm
[Sys]<Async>LinkDeinstall.fls
[Sys]<Async>LinkDeinstall.sub

This page intentionally left blank.

Number **Comment**

[Sys]<Async>LinkExample.fls
[Sys]<Async>LinkExample.sub
[Sys]<Async>LinkStart.fls
[Sys]<Async>LinkStart.sub
[Sys]<Async>LinkStop.fls
[Sys]<Async>LinkStop.sub
[Sys]<Async>ReadMe
[Sys]<Async>RequestEx.txt
[Sys]<Async>Start.c
[Sys]<Async>Stop.c

If at least one of the control files contained an entry for *OverwriteNoFiles*, then those would also have to be restored by the statements:

```
OverwriteNo=ConcatStrings(ScrSpec,
    'InstallOverwriteNoFrom.fls')
Ver = FileVersion(OverwriteNo)
If Ver = '255' Then
    Command Restore Archive  ,&
            DeviceFrom       ,&
            @[Scr]<$>InstallOverwriteNoFrom.fls,&
            @[Scr]<$>InstallOverwriteNoTo.fls,&
            No
EndIf
```

If there are no entries for *OverwriteNoFiles* in the control files of the chosen subpackages, then *InstallOverwriteNoFrom.fls* and *InstallOverwriteNoTo.fls* are not created. Therefore, we have to check for their existence. A value of '255' from FileVersion() says that the file exists but doesn't have a version (only run files have versions.) If the files did not exist and the **Restore Archive** command was issued anyway, the second and third parameters would be null. That means that the default of <*>* would be used, which is clearly not what is desired.

Number	Script

```
  6    pkg = GetMsg(6)
       If substring(pkg,pkgs) <> 65535
           Msg = GetMsg(1)
           Display(Msg)

  7        Command List Request Set ,&
                      RqSpec         ,&
                      RqListSpec

           Displayline(DoneMsg)
           DisplayLine
           Msg = GetMsg(2)
           Display(Msg)

  8    SaveVidLevel = VideoLevel
       If VideoLevel = TotalSilence
           EchoSome
       EndIf

       Command Make Request Set      ,&
                      (RqListSpec, RqTextSpec),&
                      RqSpec         ,&
                                     ,&
                      12.1.0

If SaveVidLevel = TotalSilence
       EchoOff
EndIf

           Displayline(DoneMsg)
           DisplayLine
```

Number	Comment
6.	Determine whether the Asynchronous Examples subpackage has been chosen. If so, print the message "Listing requests . . ." Then perform steps 7, 8, and 9 (**List Request Set, Make Request Set, Merge Command Files**).
7.	List the request set: *RqSpec is [Sys]<Sys>Request.sys* if the installation is private or *[!Sys]<Sys>Request.sys* if the installation is public. RqListSpec is [Scr]<$>LRS.
8.	Merge the new requests into the user's request file. Note that video must be turned on, since there might be errors reported by **Make Request Set**. If it reports errors, it stops for user input. If video were off (non-verbose), the installation would appear to hang, since the user would never see the messages. RqListSpec and RqSpec are unchanged. RqTextSpec is *[Sys]<Async>Request.Ex.txt* if the installation is private or *[!Sys]<Async>Request.Ex.txt* if the installation is public.

Number	Script

```
 9        pkgCmdName = GetMsg(7)
          CmdSpec = ConcatStrings(ScrSpec, pkgCmdName)
          Command Merge Command Files,&
                    CmdSpec         ,&
                    CmdFileTo       ,&
                    Yes             ,&
                    VolumeTo
      EndIf

      RestartLabel

10    pkg = GetMsg(8)
      If substring(pkg,pkgs) <> 65535
          pkgCmdName = GetMsg(9)
          CmdSpec = ConcatStrings(ScrSpec, pkgCmdName)
          Command Merge Command Files,&
                    CmdSpec         ,&
                    CmdFileTo       ,&
                    Yes             ,&
                    VolumeTo
      EndIf
      End
```

Number	Comment
9.	Merge the command files. CmdFileTo is *[Sys]<Sys>Sys.cmds* if the installation is private, *[!Sys]<Sys>Cluster.cmds* if it is public. VolumeTo is [Sys] if the installation is private, [!Sys] if it is public. Note that the names of the command files must come from the message file, since there is no way for the Installation Manager to pass them all to Batch.
10.	If the Development Run Files subpackage was chosen, then merge its commands.

6

Using the System Log File

Introduction

The System Log File, *Log.sys*, is created when you initialize a disk with the **Format Disk** command. Any application can read or write to the Log file, but it is used primarily by the operating system and by system services to log significant events. For example, the file system logs all disk errors in the system Log file. End users can view the Log file's contents by using the Executive's **PLog** utility. For details on **Format Disk** and **PLog**, see the *CTOS Executive Reference Manual*.

Only certain types of applications would find the Log file useful. For example, a system monitor program might check the log file periodically and send mail to a remote administrator if it finds any error messages there.

This section describes the internal format of the Log file and explains how applications can use it. The section assumes you understand how to use the file management operations. For background, you should read "File Management" in the *CTOS Operating System Concepts Manual*.

Log File Format

The contents of the system Log file are called records. There are a number of different record types for the display of different kinds of information. Each record type, however, has the same 31-byte header format, which is automatically generated by the operating system. One of the fields in the header contains the record size. This field is mentioned several times in this section because of its significance in reading the Log file. The record trailer is the type-specific information of the record. The length and organization of the trailer varies with the type of record. The Log file format as well as the different record types are described in

detail in "System Structures" in the *CTOS Procedural Interface Reference Manual*. To read the Log file, you should be familiar with these formats.

The size the Log file is set at volume initialization and can only be changed by reinitializing. For this reason, if the file gets full, the operating system writes new records in circular fashion, starting over again at the beginning of the first file sector. In this way, when the file becomes full, the oldest information is overwritten with the newest information.

Writing Messages to the Log File

The operating system and utilities write different types of records to *Log.sys*. The application program, however, can only write an ASCII record type, and this must be done using the WriteLog request. The file system is the only program that writes to the Log file. It acts as the intermediary by serving WriteLog requests and doing the writing to the Log file. For an example of how to use WriteLog, see "Error Handling Conventions" in *CTOS/Open Programming Practices and Standards*. The WriteLog request itself is described in detail in the *CTOS Procedural Interface Reference Manual*.

Displaying Messages Using PLog

If you use **PLog** to display text messages you've entered in the Log file with WriteLog, **PLog** displays the header information followed by the message

```
ASCII MESSAGE
```

Following this information, the contents of your message are displayed. Because **PLog** uses video byte streams (calls WriteBsRecord) to display messages, it does not format messages.

Although you can use **PLog**, you can write your own routine to display the contents of the Log file. To do so, however, requires knowing how to read the file. Reading the file, in turn, makes more sense if you understand how the file system writes to it in the first place. Finally, to perform any of these tasks, there are a few fields in the Volume Home Block (VHB) you need be familiar with. Let's look at the structures first.

Log File Fields in the Volume Home Block

Certain fields in the Volume Home Block (VHB) are used to read from or write to the Log file. These are defined below:

lfaLogBase is the disk location where the Log file is placed when the file is created. This field is only important to the file system, which has many requests to access the disk and needs to make these requests as rapidly as possible.

cPageLog is the number of disk sectors in the Log file.

currentLogPage is the number of the current sector to be written to (a value in the range 0 through *cPageLog*−1).

currentLogByte is the next byte to be written to in the current sector (a value in the range 0 through 511).

lfaLogBase and *cPageLog* are static values determined at disk initialization. *currentLogPage* and *currentLogByte*, however, are values that change dynamically as the file system writes to the Log file.

How the File System Writes to the Log File

The file system is the only program that writes to the Log file. The way that it writes to this file should help you see the intuitive approach to reading the file, which is described later in this section.

The file system always writes to the current sector (designated by the value of *currentLogPage*). First it reads the contents of the current (512-byte) sector into a 512-byte buffer in memory. Because there may be valid records already written to the sector, it then compares the combined value of *currentLogByte* and the record size with 512. If the combined value is less or equal to 512, the new record will fit in the buffer.

Record Fits in the Buffer

If the record fits in the buffer, the file system

1. Copies the new record to the buffer starting at offset *currentLogByte*.

2. Writes the buffer to the disk sector specified by *currentLogPage*.

3. Increments the value of *currentLogByte* by the record size.

Record Does Not Fit in the Buffer

If the record doesn't fit in the buffer, the file system does not write part of the record to this buffer and the remainder to a second buffer, spanning sectors. Instead, it

1. Increments the value of *currentLogPage* by 1.

2. Initializes the buffer to zeros.

3. Places the record at the beginning of the buffer.

4. Writes the buffer to the disk sector specified by the new *currentLogPage*.

5. Sets the value of *currentLogByte* to the record size.

Incrementing *currentLogPage* by 1 means that, when the buffer is written back to disk, it is written to the next file sector. This preserves the contents of the sector that didn't have enough room for the record. Initializing the buffer to zeros not only eliminates garbage that may have been in memory but also indicates the last valid record in a sector to the Log file reader. (The significance of zeroing out the buffer to find the last record is explained in "How to Read the Log file," later in this section.)

If the new value of *currentLogPage* is greater than or equal to the number of sectors in the Log file (*cPageLog*), the file system sets *currentLogPage* to 0 before it writes the buffer to disk. When this condition occurs, the file system OR's the value of *currentLogByte* with the mask 8000h. For example, if under these circumstances *currentLogPage* is 000Ch, the file system OR's this value to 800Ch. Doing so indicates to a program reading the Log file that the records have wrapped around to the beginning of the file: the earliest chronological sector is not the first Log file sector on the disk.

Figure 6-1 shows the contents of a 6-sector Log file that has wrapped around. The letters M, T, W, TH, and F represent the days of the week to which the corresponding sectors shown were written. Sector 0 was

written on Monday. On Tuesday sectors 1 and 2 were written, and so forth. Note that Thursday's input filled up the last sector in the file. Because of this, the file system had to write Friday's input to sectors 0 and 1. When the file system wrote to sector 0, it reset *currentLogPage* to 0 and OR'd the value of *currentLogByte* with 8000h for the benefit of the Log file reader.

The figure shows that sector 2 contains the oldest information. This sector is now the *chronological* beginning of the file even though sector 0 at LFA 0 is still the beginning of the first file sector. The chronological beginning of this file starts at the beginning of the third sector or LFA (2*512).

How to Read the Log File

Any program can read the Log file. A system service might be written, for example, to read the Log file on an occasional basis, selectively looking for errors that may need to be called to the attention of a system administrator. At the request of the user, *PLog.run* reads and displays the contents of the entire file.

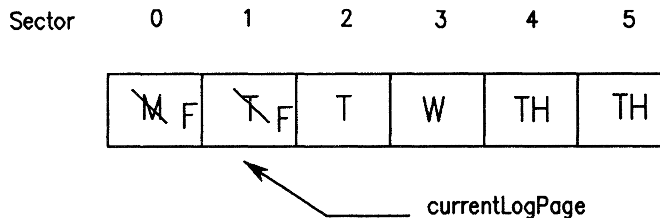


Figure 6-1. Log File Wraparound

Before reading the Log file, your program needs to gain access to the Log file fields in the VHB. Then it needs to determine whether the records in the file have wrapped around.

Accessing the Log File Fields in the VHB

Before reading the Log file, your program needs to gain access to the following fields in the VHB: *cPageLog*, *currentLogPage*, and *currentLogByte*. To do so, your program can call the GetpStructure operation to get a pointer to the VHB. (GetpStructure is described in detail in the *CTOS Procedural Interface Reference Manual*.)

Determining if Records Have Wrapped Around

To find out if the records have wrapped around so your program can tell where to start and stop reading, AND the value of *currentLogByte* with the mask 8000h. A nonzero value means the records have wrapped around, for example, ANDing with a *currentLogByte* value of 800Ch is shown below:

```
1000 0000 0000 0000     8000h mask
1000 0000 0000 1100     800Ch currentLogByte
-------------------
1000 0000 0000 0000     Result of ANDing is a nonzero value
```

Reading the File in Chronological Order

If Records Have Not Wrapped Around

If the records have not wrapped around (*currentLogByte* AND 8000h equals zero), start reading the first file sector at LFA 0 and proceed as follows:

1. Read the first record starting at offset 0 in the sector. Then process the record. (*Process* means do what you want with it. Depending on the record type, you may wish to skip the record, print it, and so forth. You can obtain general information about the record, such as its type and size from the 31-byte standard record header automatically generated when the file system wrote the record to the Log file.)

2. To read the next record, increment the offset by the size of the first record. Then check the field *Record.size* in the next record's header. If this value is 0, you have reached the end of the valid records in

this sector. This is so because when the file system sees that the next record to be written won't fit in the current sector, it writes the record to the next sector using a buffer initialized with zeros. As a result, the unused portion of a sector always contains zero values. (See "How the File System Writes to the Log File," earlier in this section.)

3. Continue processing records and incrementing the offset to advance to the next record until the value of *Record.size* equals 0.

4. Compare the number of sectors you've read so far to the value of (*currentLogPage*+1). If the numbers are equal, you are done reading all the records in the Log file. Otherwise, repeat steps 1 through 4 with the next sector until you have processed *currentLogPage*+1 sectors.

If Records Have Wrapped Around

If the records in the file have wrapped around (*currentLogByte* AND 8000h equals a nonzero value), proceed as follows:

1. To start reading from the earliest sector written, read the file beginning at sector (*currentLogPage*+1). (See Figure 6-1. In the figure, *currentLogPage* is 1. The earliest sector written is sector 2.)

2. Process the records in this sector as described in steps 1 through 3 in "If Records Have Not Wrapped Around."

3. If this sector is the last file sector [(*currentLogPage*+1) = *cPageLog*], proceed to step 4. Otherwise, continue processing the records in each sector until you have processed all records in the last file sector.

4. At the end of the last file sector, read the file starting at the beginning of the first file sector (LFA 0). Process the records as described in "If Records Have Not Wrapped Around."

Reading the File in Reverse Chronological Order

Saving Offsets to Records

Reverse chronological order is the order **PLog** uses to read and display files.

Because the records in the Log file are of variable length, the only way to find out where each record starts in a sector is to read the sector first from the beginning to the end. Record the size of each record you read. You will know when you get to the last record in the sector because *Record.size* will be 0. From the record sizes, calculate the offsets to the beginning of each record, and save these values in an array.

The next two sections present a stepwise intuitive approach to reading the records. This is followed by the algorithm **PLog** uses to process each sector.

If Records Have Not Wrapped Around

1. Start at the sector specified by *currentLogPage*.
2. Save away record offsets as described in "Saving Offsets to Records."
3. Look up and process each record in the array of record offsets, starting with the record at the last offset in the sector and proceeding to the record at offset 0.
4. Repeat steps 2 and 3 for the sector preceding the current one (*currentLogPage*–1).
5. Repeat step 4 until you have read the first sector in the file.

If Records Have Wrapped Around

1. Perform all steps described in "If Records Have Not Wrapped Around."
2. Start with the last sector in the file. (The value of *cPageLog* is the number of this sector).

3. Repeat steps 2 through 4 in "If Records Have Not Wrapped Around" until you have read the sector number (*currentLogPage*+1).

PLog's Algorithm For Processing Each Sector

To process each sector, **PLog** uses the algorithm shown below. The algorithm is shown in pseudocode. It is not intended to be compiled. The variables in the algorithm have the following meanings:

Buffer is a 512-byte array.

Lfa is the value (512 * page number currently being processed).

pRecord points to a Log File Record structure. The format of this structure is described in "Log File Record Format" in "System Structures" in the *CTOS/VM Reference Manual*.

pTRecord points to another Log File Record structure. The format of this structure is the same as *Record*.

This algorithm is used once for each sector of the log file.

```
/*   Previous Declarations
     LogFileRecordType   *pRecord
     LogFileRecordType   *pTRecord
*/

erc = Read(fhLogFile,&Buffer,512,Lfa,&cb);
pRecord = &Buffer;
pTRecord = &Buffer;

do while (pRecord->size != 0) {
     i = 0;
     do while ((pTRecord->size != 0) && (i < cb)) {
           pRecord = pTRecord;
           i = i + Record.size;
           pTRecord = &Buffer[i];
           }
     ProcessRecord(pRecord);
     pRecord->size = 0;
     }
```

Listing 6-1. PLog's Record-Processing Algorithm

7 Writing System Services for the XE-530

This chapter describes some of the issues you need to resolve if you want to write system services that run on an XE-530 with CTOS/XE 3.0 or greater. This chapter addresses some common pitfalls encountered when porting system services from 80186-based shared resource processors (SRPs). It also describes some recommended techniques to help make new system services compatible with current and future products. In general, this chapter refers to the XE-530 and its predecessors as SRPs.

Portation Issues for Existing Programs

This section describes some of the issues you may need to address if you plan to convert an existing real-mode SRP application to run in protected mode on an XE-530. You should perform this conversion if your real-mode application is a system service. The XE-530 does not support real-mode system services on GP processor boards.

If your real-mode application is not a system service, conversion is recommended, but optional. Cleanly-written application programs that run either on a workstation or on an 80186-based SRP should run unaltered on an XE-530. However, converting your application to run in protected mode has many advantages, among them the protection mechanism itself and access to vastly greater amounts of memory.

General Guidelines

The single most important rule to follow when writing programs for the XE-530 is that those programs must conform to the general guidelines for protected mode programming. These guidelines can be found in "Protected Mode Programming Guidelines" in the *CTOS/Open Programming Practices and Standards* manual.

If your application conforms to the guidelines listed in that chapter, it may require only relinking to run in protected mode.

The other general requirement for existing SRP applications is to remove any direct memory references to operating system data structures. These structures may have been relocated in the new operating system release. You should try to avoid any direct use of operating system data structures. If you must have access to one, use the GetpStructure operation with the appropriate case value.

For most applications and system services, conforming to the rules described in the paragraphs above will allow them to run in protected mode on an XE-530.

The two sections below describe unsupported mechanisms used illicitly by some real-mode SRP software. These mechanisms do not work on CTOS/XE 3.0 or greater. If your software uses them, convert it to use a supported interface.

User Numbers and Exchanges

In real-mode SRP operating systems, exchanges and user numbers were assigned by the OS in the range from 1 to 255. Some programs took advantage of this fact, using the high-order byte of the UserNum and/or ExchResp fields of a request block for their own purposes.

In CTOS/XE 3.0, the operating system uses the full word size of these fields. Any program which uses these fields to contain anything other than their true user number and response exchange will fail.

Remote Memory and Inter-CPU Communication

On older versions of SRP operating system it was possible (though not recommended) for a program to bypass the Inter-CPU Communication mechanism and write directly to the memory of a remote processor. This is no longer possible with CTOS/XE 3.0 or greater.

The internal mechanism by which processor boards communicate has changed substantially, and existing programs that attempt to use the old mechanism will fail.

If a program needs to communicate with a remote processor, it should use the operating system Request interface. The performance of the Request interface with CTOS/XE 3.0 or greater has been substantially improved over previous SRP operating system versions, even on existing hardware. Programs should no longer have any legitimate need to bypass the Request interface.

Inter-CPU Communication Buffer Block Size Issues

In older SRP operating systems, two types of Inter-CPU Communication buffer existed: the Z-block and the Y-block. The Z-block was used for small (less than 180 byte) request buffers and the Y-block was used for larger (up to 2560 byte) request buffers.

CTOS/XE 3.0 includes these types and a new one, the W-block. The W-block can accommodate very large request buffers (at least 5120 bytes). However, *the inclusion of W-blocks in the operating system is a configurable parameter*. This means that user-customized SRP operating systems may not have any.

For this reason, requests that may require a buffer of more than 2560 bytes should be structured so that they can be piecemealed. This ensures that the request can be used in any hardware configuration.

For instructions on how to make a request piecemealable, see "Writing Request-Based System Services" in the *CTOS/Open Programming Practices and Standards* manual.

The Demise of the MCommands

With older SRP operating systems, a user or administrator entered commands which began with the letter M to run utilities on the SRP. These MCommands no longer exist with Standard Software 12.0. They, and the Administrative Agent software, have been replaced by ClusterView.

See the *CTOS System Administration Guide* for more information about the ClusterView software.

Restriction on the GetWSUserName Operation

On a workstation server, the GetWSUserName operation can be used to obtain the user name of any cluster user. On the SRP, however, the GetWSUserName operation should not be used. At best, it will retrieve only those user names connected to the local processor. There is no clean method of identifying the users of a system service on an SRP.

Controlling the Routing of Requests on the XE-530

This section describes the inter-board request routing methods supported by CTOS/XE 3.0. System services which run on the SRP must define some form of inter-board routing for the requests they serve. This section should help the system service writer to decide which routing method is appropriate for their service.

The SRP Request Routing Directives

Several types of SRP inter-board routing are supported. Each is listed, with a short description, in Table 7-1.

The sections below describe the three most commonly used routing types in greater detail. Almost all system services should use one of these three routing types. The other types are intended for rare special cases, and for internal use by the operating system.

Table 7-1. SRP Request Routing Types

(Page 1 of 2)

Field	Description
rLocal*	If the request (block) does not contain a network routing specification for a remote board, the request is served locally. Unless the request is served locally, the client receives status code 33 ("Service not available"). If the request contains a specification for a remote board, the SRP filter process calls RequestRemote to route the request to the board specified in the *master FP Name table*. The filter uses the specification name (such as Win1 or FP00) in square brackets as the key to locate the corresponding board's slot number in the table. The specification must contain a name in square brackets or an error code is returned, terminating routing.
rRemote*	If the request is served locally, *rRemote* is the same as *rLocal*. Otherwise the request is sent by exchange to the remote board. When a system service calls ServeRq during installation, ServeRq updates the operating system request routing table on each board to reflect the system service's slot number. This means that only one system service on an entire SRP can serve a request that is routed rRemote.
rDevice*	If the request contains neither a handle nor specification (no network routing), the request is routed to all boards previously accessed by this user number. An example of such a request might be a termination request. If the request contains a handle (network routing by file handle), it is routed by the handle to the appropriate board. See the discussion of handles, later in this chapter, for more information. If the request contains a specification for a remote board (network routing by device), the SRP filter process calls RequestRemote to route the request to the board specified in the *master FP Name table*. The filter uses the specification name (such as Win1 or FP00) in square brackets as the key to locate the corresponding board slot number in the table. If the specification does not contain a name in square brackets, the Kernel on this board attempts to send the request to a local system service.
rBroadcast	The request is routed to every processor board booted on the SRP.

*This type is frequently used.

Table 7-1. SRP Request Routing Types
(Page 2 of 2)

Field	Description
rMasterFP	The request is routed to the Master FP.
rFileId	The request is routed to the appropriate board by the slot number contained in the first byte of the request block control information.
rLineNum	The request is routed to the Cluster Processor (CP) that handles this line. This routing type is used by the operation MegaFrameDisableCluster. Each CP has two lines. For example, CP00 has lines 1 and 2; CP01 has lines 3 and 4; and so on. (For details, see Chapter 43, "Cluster Management").
rHandle	The request is routed to the target file processor (such as an FP, DP, or GP+SI) using an indexed field in the handle.
rMasterCp	(Unused)

*This type is frequently used.

Local Routing

Local routing (rLocal) is the simplest form of SRP routing. In general, this type of routing is used when a request should be served on the board where it was generated.

When a system service uses local routing for its requests, it must be installed on the same processor as its clients. If its clients are cluster workstations, it must be installed on the processor to which those cluster workstations are physically connected.

If a system service uses local routing, multiple instances of it can be installed on an XE, but only local clients can communicate with each one. In other words, if a user installs the system service on each of two CP processors, users on each CP can talk to their local copy of the system service. However, users on each CP cannot communicate with the copy

of the service on the other CP. Likewise, if the system service is installed on an FP, only other programs running on that FP can communicate with it.

Because of these restrictions that local routing places on the location of the system service within the XE, this routing type should be used sparingly.

Remote Routing

Remote routing (rRemote) is the most commonly used type of SRP request routing, and the simplest for both the programmer and the user. Remote routing allows one instance of a system service to serve clients on all processor boards and all cluster workstations. These clients need not know where the system service resides. The request is routed automatically to the correct processor board by the operating system.

The single restriction imposed by the use of remote routing is that only one instance of the system service can be installed on the XE. For most system services, however, only one instance is needed.

Listing 7-1 shows a sample *Request.txt* file for a system service that uses remote routing. The sample system service described in *CTOS/Open Programming Practices and Standards* runs unaltered on the XE-530 if you use the Request.txt file shown in Listing 7-1.

```
:WsAbortRq:      0EF05h
:TerminationRq:  0EF05h
:SwappingRq:     0EF04h
:ChgUserNumRq:   0EF06h

:RequestCode:    0EF00h
:RequestName:    GetFooText
:Version:        3
:LclSvcCode:     0000h
:ServiceExch:    exchInstalledMastr
:sCntInfo:       2
:nReqPbCb:       0
:nRespPbcb:      2
:Params:         w(12), p(14), w(18), p(20), c(2,24)
:NetRouting:     rFh
:SrpRouting:     rRemote

:RequestCode:    0EF01h
:RequestName:    DeinstallFooServer
:Version:        2
:LclSvcCode:     0000h
:ServiceExch:    exchInstalledMastr
:sCntInfo:       0
:nReqPbCb:       0
:nRespPbcb:      0
:Params:         none
:NetRouting:     noRouting
:SrpRouting:     rRemote
```

Listing 7-1. Sample Request.txt File Using Remote Routing (Page 1 of 3)

```
:RequestCode:     0EF02h
:RequestName:     OpenFooServer
:Version:         2
:LclSvcCode:      0000h
:ServiceExch:     exchInstalledMastr
:sCntInfo:        0
:nReqPbCb:        0
:nRespPbcb:       1
:Params:          p(12), c(2,16)
:NetRouting:      OpenFh,CloseAtTermination
:SrpRouting:      rRemote

:RequestCode:     0EF03h
:RequestName:     CloseFooServer
:Version:         2
:LclSvcCode:      0000h
:ServiceExch:     exchInstalledMastr
:sCntInfo:        2
:nReqPbCb:        0
:nRespPbcb:       0
:Params:          w(12)
:NetRouting:      CloseFh,rFh
:SrpRouting:      rRemote

:RequestCode:     0EF04h
:RequestName:     SwapClientFooServer
:Version:         2
:LclSvcCode:      0000h
:ServiceExch:     exchInstalledMastr
:sCntInfo:        2
:nReqPbCb:        0
:nRespPbcb:       0
:Params:          none
:NetRouting:      noRouting
:SrpRouting:      rRemote
```

Listing 7-1. Sample Request.txt File Using Remote Routing (Page 2 of 3)

```
:RequestCode:    0EF05h
:RequestName:    TerminateClientFooServer
:Version:        2
:LclSvcCode:     0000h
:ServiceExch:    exchInstalledMastr
:sCntInfo:       2
:nReqPbCb:       0
:nRespPbcb:      0
:Params:         none
:NetRouting:     noRouting
:SrpRouting:     rRemote

:RequestCode:    0EF06h
:RequestName:    ChangeUserClientFooServer
:Version:        2
:LclSvcCode:     0000h
:ServiceExch:    exchInstalledMastr
:sCntInfo:       2
:nReqPbCb:       0
:nRespPbcb:      0
:Params:         none
:NetRouting:     noRouting
:SrpRouting:     rRemote
```

Listing 7-1. Sample Request.txt File Using Remote Routing (Page 3 of 3)

Routing by Device Specification

Routing by device specification (rDevice) allows multiple instances of the same system service to be installed on different processors in an XE. For example, routing by device specification allows an SNA Network Gateway to be installed on each of two GP/CI boards. Cluster workstation clients can then specify which of the SNA Network Gateways they want to use, by naming the processor board on which that Gateway is installed.

Routing by device specification is most useful for connection-oriented resource-controlling programs. An SNA Network Gateway is an example of such a program, since it controls access to an external computer

network and its clients are terminal emulation programs, which establish a communications session with a remote computer on the SNA network.

Listing 7-2 shows a sample *Request.txt* file for a system service that uses routing by device specification.

The sample system service described in *CTOS/Open Programming Practices and Standards* runs unaltered on the XE-530 if you use the *Request.txt* file shown in Listing 7-2. However, you need to change its client programs.

Listing 7-3 shows a client program that can access the system service on any processor board in the XE-530. The user simply specifies the board name (for example, "GP00") as a runtime parameter to the client program.

Finally, Listing 7-4 shows two enhancements you can make to the sample system service in *CTOS/Open Programming Practices and Standards*. These changes cause the system service to identify the name of the board on which it is running, and to return that name as part of the message to its clients.

```
:WsAbortRq:       0EF05h
:TerminationRq:   0EF05h
:SwappingRq:      0EF04h
:ChgUserNumRq:    0EF06h

:RequestCode:     0EF00h
:RequestName:     GetFooText
:Version:         4
:LclSvcCode:      0000h
:ServiceExch:     exchInstalledMastr
:sCntInfo:        2
:nReqPbCb:        0
:nRespPbcb:       2
:Params:          w(12), p(14), w(18), p(20), c(2,24)
:NetRouting:      rFh
:SrpRouting:      rDevice
```

Listing 7-2. Sample Request.txt File Using Device Routing (Page 1 of 3)

```
:RequestCode:    0EF01h
:RequestName:    DeinstallFooServer
:Version:        2
:LclSvcCode:     0000h
:ServiceExch:    exchInstalledMastr
:sCntInfo:       0
:nReqPbCb:       0
:nRespPbcb:      0
:Params:         none
:NetRouting:     NoRouting
:SrpRouting:     rRemote

:RequestCode:    0EF02h
:RequestName:    OpenFooServer
:Version:        3
:LclSvcCode:     0000h
:ServiceExch:    exchInstalledMastr
:sCntInfo:       6
:nReqPbCb:       1
:nRespPbcb:      1
:Params:         pbcb0, p(24), c(2,28)
:NetRouting:     DevSpec,OpenFh,CloseAtTermination
:SrpRouting:     rDevice

:RequestCode:    0EF03h
:RequestName:    CloseFooServer
:Version:        3
:LclSvcCode:     0000h
:ServiceExch:    exchInstalledMastr
:sCntInfo:       2
:nReqPbCb:       0
:nRespPbcb:      0
:Params:         w(12)
:NetRouting:     CloseFh,rFh
:SrpRouting:     rDevice
```

Listing 7-2. Sample Request.txt File Using Device Routing (Page 2 of 3)

```
:RequestCode:    0EF04h
:RequestName:    SwapClientFooServer
:Version:        2
:LclSvcCode:     0000h
:ServiceExch:    exchInstalledMastr
:sCntInfo:       2
:nReqPbCb:       0
:nRespPbcb:      0
:Params:         none
:NetRouting:     noRouting
:SrpRouting:     rDevice

:RequestCode:    0EF05h
:RequestName:    TerminateClientFooServer
:Version:        2
:LclSvcCode:     0000h
:ServiceExch:    exchInstalledMastr
:sCntInfo:       2
:nReqPbCb:       0
:nRespPbcb:      0
:Params:         none
:NetRouting:     noRouting
:SrpRouting:     rDevice

:RequestCode:    0EF06h
:RequestName:    ChangeUserClientFooServer
:Version:        2
:LclSvcCode:     0000h
:ServiceExch:    exchInstalledMastr
:sCntInfo:       2
:nReqPbCb:       0
:nRespPbcb:      0
:Params:         none
:NetRouting:     noRouting
:SrpRouting:     rDevice
```

Listing 7-2. Sample Request.txt File Using Device Routing (Page 3 of 3)

```
#define CheckErc
#define Delay
#define ErrorExit
#define OutputToVid0
#define ReadKbdDirect
#define Syslit
#include <ctoslib.h>
#include <stdlib.h>
#include <string.h>

pragma Calling_convention (CTOS_CALLING_CONVENTIONS);
extern ErcType CloseFooServer(Word wHandle);
extern ErcType GetFooText(Word wHandle, Pointer
                          pbDataRet, Word cbDataRet,
                          Pointer psDataRet);
extern ErcType OpenFooServer (Pointer pbDevSpec, Word
                          cbDevSpec, Pointer pHandleRet);
pragma Calling_convention ();

#define CHECKONLY 1
#define DELAYRATE 10
#define ERCNOCHARAVAIL 602

Word hConnect;
Word cbRet;

char bKey;
char rgbServerData[80];

char pMsgIntro [] =
"Press any key to close the connection with service.\n";
char *pbDevSpec = NULL;
Word cbDevSpec = 0;
```

Listing 7-3. A Client Program that Supports Device Routing (Page 1 of 2)

```
void main (int argc, void *argv[])
   {
   /* get device spec from user param */
   if(argc > 1) {
      pbDevSpec = argv[1];
      cbDevSpec = strlen(pbDevSpec);
      }

   /* open my connection with the foo server */
   CheckErc( OpenFooServer(pbDevSpec, cbDevSpec,
             &hConnect));

   /* display the intro message */
   OutputToVid0 (pMsgIntro, strlen(pMsgIntro));

   /* loop checking for a keyboard character, any one */
   while(ReadKbdDirect(CHECKONLY,&bKey)==ERCNOCHARAVAIL)
      {
      /* get a message from the server */
      CheckErc (GetFooText (hConnect, &rgbServerData,
                            sizeof(rgbServerData), &cbRet));
      /* display the message */
      OutputToVid0 (&rgbServerData, cbRet);
      /* now wait awhile before continuing */
      CheckErc (Delay (DELAYRATE));
      }

   /* close my connection */
   CheckErc (CloseFooServer (hConnect));
   ErrorExit (0);
   }
```

Listing 7-3. A Client Program that Supports Device Routing (Page 2 of 2)

```
/* new version of ProcessDataRequest returns board name
 */
void ProcessDataRequest (RqDataType *pRq)
   {
   int  i = 0;
   Word wHandle = pRq->wHandle;

   /* check for valid handle */
   if (pRq->RqHead.userNum != rgHandles[(wHandle)])
      {
      pRq->RqHead.ercRet = ERCINVALIDHANDLE;
      return;
      }
   /* calculate the maximum string size */
   if (pRq->cbDataRet < strlen(pMsgRet) +
       strlen(rgbDeviceId))
      i = pRq->cbDataRet;
   else
      i = strlen (pMsgRet) + strlen(rgbDeviceId);

   /* put the message and size in the user's buffer */
   *((char *) pRq->pbDataRet) = 0;
   strncpy (pRq->pbDataRet, pMsgRet, i);

   /* attach device spec */
   if (i-strlen(pMsgRet) > 0)
      strncat(pRq->pbDataRet, rgbDeviceId,
              i-strlen(pMsgRet));
   *((Word *)(pRq->psDataRet)) = i;

   /* return an error OK condition */
   pRq->RqHead.ercRet = 0;
   }
```

Listing 7-4. Changes to Sample System Service to Illustrate Device Routing (Page 1 of 2)

```
/*
 *  The following code comes later, inside of main()
 *  function for program, but before ConvertToSys.
 */

   /* see if I'm running on a master */
   /* get config block */
   CheckErc(GetpStructure(0x2C8, 0, &pMySysConfigRet));
   /* check offset 33 */
   if(pMySysConfigRet->ClusterConfiguration == 2)
      fMaster = TRUE;
   else
      fMaster = FALSE;
   /* see what my board ID is if I'm on an SRP */
   if(fMaster) {
      erc = GETPROCINFO(&i, &i, &i, rgbDeviceId,
                        DEVICEIDMAX, &cbDeviceId);
      if(erc == ercOK)
         /* null terminate the string */
         rgbDeviceId[cbDeviceId] = 0;
      else
         /* null string */
         rgbDeviceId[0] = 0;
      }
```

Listing 7-4. Changes to Sample System Service to Illustrate Device Routing (Page 2 of 2)

Other Routing Methods

The other SRP request routing methods are rarely used, except by the operating system itself. However, their use is not discouraged. If your application needs to use one of them, you should use it. See the *CTOS Operating System Concepts Manual* for a general description of SRP routing.

Bear in mind, however, that some of the rarely-used SRP routing types require specially formatted connection handles. These requirements are described below.

Use of Handles on the XE-530

On the SRP, most request routing types use the same format for handles as is used on a workstation. Because almost all system services use these routing types, system services for the SRP rarely need to use special handles.

The following SRP routing types use standard connection handles:

rLocal
rRemote
rDevice
rBroadcast

The following SRP routing types require special handles:

rHandle
rFileId
rLineNumber

The formats of these special handles are described later in this section.

The Standard Connection Handle

Most SRP routing types use standard connection handles. These handles have the following format:

Bit 15: reserved for network use.

Bits 13 and 14: set to 1s if system service is installed on a server.

Bit 12: Reserved. Should be set to 0.

Bits 11 to 0: Available for use by the system service.

Note, however, that the system service and its client should not attach meaning to any bits within the handle. The service cannot pass information to the client by returning a particular value in the connection handle. CTOS guarantees that both the service and the client will perceive a unique handle value, but *that value may not be the same* for both the client and the service.

Non-Standard Handle Types

Requests that are routed rHandle, rFileId, or rLineNumber require specially formatted handles. The format for each type is described below.

rHandle: Connection handles for requests that are routed rHandle should have the following format:

Bit(s)	Description
13 through 15	Standard.
10,11,12	FPIndex. See description below.
0 through 9	Available for use by service.

The FPIndex field contains the index of the FP (or GP/SI) board on which the service is installed. In other words, if the service is installed on FP00, the FPIndex is 0. If the service is installed on FP01, the FPIndex is 1.

rFileId: Connection handles for requests that are routed rFileId should have the following format:

Bit(s)	Description
12 through 15	Standard.
8 through 11	Available for use by the service (4 bits).
0 through 7	Processor Slot ID (for example, 77h)

rLineNumber: Connection handles for requests that are routed rLineNumber should have the following format:

Bit(s)	Description
12 through 15	Standard.
8 through 11	Available for use by the service (4 bits).
0 through 7	Cluster Line Number. See description below.

The cluster line number is a unique cluster line index for the entire SRP starting at cluster line 1. For example, CP00 (or GP00) owns lines 1 and 2, CP01 (or GP01) owns lines 3 and 4, and so on.

8

The Synchronous CommLine Interface

This chapter describes extensions to the CommLine interface for synchronous data communications.

The CommLine Interface

The CommLine interface is discussed in detail in "Communications Programming" in *CTOS/Open Programming Practices and Standards*. That chapter explains the components of the interface and gives general guidelines for using it. The chapter also gives general examples of initializing the serial controller and includes a sample communications interrupt service routine. Unless you already understand the CommLine interface, read that chapter before continuing.

To recap the information in *CTOS/Open Programming Practices and Standards*, the CommLine interface is a fairly low-level interface used for synchronous data communications. The interface is designed to provide as much device-independence as possible without sacrificing performance. The CommLine interface allows a program that uses it to operate on any workstation, using any serial port.

However, because performance was the primary consideration in its design, the CommLine interface requires the programmer to perform a certain amount of serial controller initialization via direct register writes.

Extensions to the Traditional CommLine Interface

There were few choices to make in the traditional CommLine interface. The program called InitCommLine, and received back two port addresses, a data port and a control port. The program used the data port to exchange words of user data with the serial controller. The

program used the control port to perform any initialization or control that required direct reads or writes to the registers on the serial controller.

The additional operations ReadCommLineStatus and WriteCommLineStatus controlled RS-232 signals which were not controlled directly by 8274-type serial controllers. This allowed the application to query and set these signals in a device-independent fashion.

Lastly, the ChangeCommLineBaudRate operation allowed the application to set the speed of data transfer in a device-independent way.

All these calls remain in the current version of the interface, but extensions have been added. The subsections below address each operation in turn.

InitCommLine

InitCommLine has undergone substantial changes from its traditional form for the CTOS/XE 3.0 and later versions of the operating system. However, these changes have been made in such a way that it remains compatible with existing programs. The mechanism by which this is accomplished is as follows.

When a new feature is added to the available hardware, a new field is added to the Communications Line Configuration Block and to the Communications Line Return Area for subsequent versions of the operating system.

Programs which know of and desire to use this feature can set that field in the Communications Line Configuration Block to some specific nonzero value. If the desired feature is present in the current hardware, the operating system sets the corresponding field in the Communications Line Return Area to a nonzero value.

This way, the program knows that the desired feature is present in hardware and supported by the operating system. If the feature is not available, the operating system returns zero in the appropriate field of the Communications Line Return Area. The program can then continue or terminate, as appropriate for its application. Note that the program must initialize the Communications Line Return Area to all zeros for this mechanism to work correctly.

The operating system is also able to identify which features a program does not know about. This is because the program specifies the size of the Communications Line Configuration Block and of the CommLine Return Area in its call to InitCommLine. Any fields beyond the specified size are unknown to the program, and therefore clearly not used by it.

Differences Between 8274 and 82530 Communication Controllers

The 8274-type and 82530-type serial communication controllers are very similar, but they are not quite register compatible. This has at least some effect on almost all synchronous communications programs. The table below shows which registers differ between the two controllers.

Table 8-1. 8274 and 82530 Register Differences

Register No. and Bit Position*	8274	82530
RR0, bit 1	Interrupt Pending	Zero Count
WR0, bits 3-5 = 001	Send Abort	Point High (access regs 9-15)
WR0, bits 3-5 = 011	Channel Reset	Send Abort
WR1, bit 2	Status Affects Vector	Parity is Special Condition
WR1, bits 3-4 = 00	RxInt/DMA Disable	RxInt Disable
WR1, bits 3-4 = 10	Int. on Rx, Special, and Parity	Int. on Rx and Special
WR1, bits 3-4 = 11	Int. on Rx and Special	Int. on Special Only
WR1, bit 6	Always 0	Ready/DMA Req. Function
WR2 (channel A only)	Interrupt Control Functions	Channel A Interrupt Vector
RR3	N/A	Similar to Interrupt Pending
WR9, bit 0	N/A	Status Affects Vector
WR9, bits 1-5	N/A	Interrupt Control Functions
WR9, bits 6-7	N/A	Channel Reset

* Bits are numbered 76543210

ChangeCommLineBaudRate

ChangeCommLineBaudRate allows a program to modify a synchronous channel's data rate dynamically.

8274-type serial controllers are not affected by this operation, though errors may occur if a data transfer is in progress when this operation is called.

If the serial controller is 82530-type controller, this operation sets the serial controller to the new data rate. However, this operation does not change the clock source. If a program wants to change from internal to external clocking, it must do so via direct serial controller port I/O.

Lastly, on channels which do not support separate transmit and receive baud rates, the value of the *iRxTx* parameter to the ChangeCommLineBaudRate operation must be 0 ("both"). If a program attempts to set the receive and transmit baud rates separately on hardware that does not support that feature, an error code (7 or 61) is returned and the operation is not performed.

ReadCommLineStatus

ReadCommLineStatus performs the same functions as before, but has additional case values. ReadCommLineStatus can now read the following signals:

Secondary Receive Data (SRxD)
Data Set Ready (DSR)
Ring Indicator (RI)
Secondary Carrier Detect (SCD)
Secondary Clear-to-Send (SCTS)
Signal Quality

Note that not all hardware supports the SCD, SCTS and Signal Quality signals. See your hardware documentation for more information.

WriteCommLineStatus

Like ReadCommLineStatus, WriteCommLineStatus is essentially unchanged but supports additional case values. WriteCommLineStatus can now set or clear the following signals:

Secondary Receive Data (SRxD)
Secondary Request to Send (SRTS)
Rate Select
Select Standby
Data Terminal Ready (DTR)

Programs that use DMA *must* use WriteCommLineStatus to set or clear the DTR signal. Programs which do not use DMA must do so using direct serial controller port I/O.

Using DMA with Synchronous Data Communication

CTOS now provides a device-independent interface for programs that want to use DMA for communications I/O. DMA for serial communications is only supported on certain hardware, such as the B39 (386i) and the GP processor on the XE-530. (See your hardware documentation for more specific information.) Programs can transfer blocks of data as large as 64K bytes using DMA, though a smaller block size is more commonly used.

This section describes how to use the CommLine DMA interface, which includes the following system common procedures:

TransmitCommLineDMA (alias: XmitCommLineDMA)
ReceiveCommLineDMA (alias: RecvCommLineDMA)
GetCommLineDMAStatus

Note that because the CommLine DMA operations are system common procedures, they can be called directly from interrupt service routines.

Initializing Communications DMA

If a communications program wants to use DMA, it must first specify that fact in its call to InitCommLine.

The Communications Line Configuration Block contains a field, *fDMA*, which allows the program to indicate that it wants to use DMA for communications I/O. To request DMA, the program must set the value of this field to 0FFh (255).

After it calls InitCommLine, the program must check the value of the *DMAavailable* field of the Communications Line Return Area. If the value is 0FFh (255), DMA hardware is available for the program to use. If the value is 0, no DMA hardware is available.

If communications DMA hardware is present, InitCommLine sets the serial controller to use DMA I/O mode instead of interrupt I/O mode. This requires setting certain bits in write register 1 (bits D7, D6, D5) and write register 14 (bit D2) of the serial controller. If a DMA-using program writes to either of these registers, it must be careful to preserve the settings of the DMA mode control bits.

In addition, if a DMA-using program needs to assert or de-assert the RS-232 DTR signal, it must use the WriteCommLineStatus procedure to do so.

Finally, note that when DMA is requested, InitCommLine performs its normal functions plus setting up the serial controller to use DMA, but nothing more. The client communications program is still responsible for completing the initialization of the serial controller.

TransmitCommLineDMA

The TransmitCommLineDMA procedure (called XmitCommLineDMA in BTOS 3.x) sets up the DMA controller to transfer data directly from memory to the serial communication controller. This procedure returns immediately, since it merely sets up a future data transfer. It does not wait for the data transfer to occur.

The TransmitCommLineDMA procedure takes three parameters: the CommLine connection handle, a pointer to a data buffer and the size of the buffer. After TransmitCommLineDMA returns to its caller, the DMA controller transfers this data one byte at a time from the specified buffer as the serial controller requests it. The transfers continue until the entire buffer has been sent. For efficient use of the system data bus, the buffer should be either word-aligned or doubleword-aligned, never byte-aligned.

When the specified number of bytes have been sent, the serial controller sends the CRC check value, then generates an External/Status interrupt. The program's External/Status ISR should verify that the correct number of bytes was transmitted, then call TransmitCommLineDMA with the next buffer of data. The serial controller resumes DMA transfers before the TransmitCommLineDMA call returns.

ReceiveCommLineDMA

The ReceiveCommLineDMA procedure (called RecvCommLineDMA in BTOS 3.x) sets up the DMA controller to transfer data directly from the serial communication controller to memory. It is the reverse of the TransmitCommLineDMA procedure. Like the transmit procedure, ReceiveCommLineDMA returns immediately, since it merely sets up a future data transfer. It does not wait for the data transfer to occur.

The ReceiveCommLineDMA procedure takes four parameters: the CommLine connection handle, a pointer to a data buffer, the size of the buffer and a pointer to a returned count of bytes transferred. After ReceiveCommLineDMA returns to its caller, the DMA controller transfers data one byte at a time to the specified memory buffer as the serial controller receives it. The transfers continue either until a complete data frame is received, or until the transfer count reaches the size of the buffer. As with transmit buffers, the receive buffer should be word-aligned or doubleword-aligned, but never byte-aligned.

When the serial controller detects the end of a data frame (EOF) or a full buffer, it generates a Receive Special interrupt. The program's receive ISR should verify that a complete frame was received successfully, then call ReceiveCommLineDMA with the next available buffer for data.

Note that the 82530 controller does not indicate EOF until after all characters have been read from its buffer. If the DMA receive buffer is too small, EOF may be detected but never reported. For this reason, it is important to ensure that the receive buffer is slightly larger than the largest possible incoming frame.

The ReceiveCommLineDMA procedure also returns the number of bytes successfully received into the previous buffer. This allows the program to know how much valid data is present in the buffer without making any additional procedure calls.

The serial controller resumes DMA transfers before the ReceiveCommLineDMA call returns.

GetCommLineDMAStatus

The GetCommLineDMAStatus procedure returns the count of bytes transferred for the current DMA operation, in both the transmit and the receive direction.

If called from the External Status ISR, it allows the program to determine whether a transmission was completed successfully. To do so, the program simply compares the returned byte count to the size of the transmit buffer. If they are equal, all bytes were sent successfully.

If called from the Receive Special ISR, it allows the program to determine the size of the data in the receive buffer. While the ReceiveCommLineDMA procedure also provides this information, some programs may need to use GetCommLineDMAStatus anyway. For example, if the program needs to verify the received data in some way before requesting the next transfer, it should determine the size of the received data by calling GetCommLineDMAStatus.

Implementing X.21 (1984) Protocol Support

Certain hardware models, namely the XE-530 GP/CI processor set, include special hardware which supports the X.21 signalling protocol. This section describes the interface to that hardware, and the programming tasks required to use it.

What Is the X.21 Protocol?

X.21 is a digital signalling protocol defined by the CCITT (Consultative Committee for International Telegraph and Telephone). It allows a DTE (Data Terminal Equipment) to set up and clear data calls by exchanging signals with the DCE (Data Circuit-Terminating Equipment) connected to it. The X.21 protocol uses a combination of data and control signals to accomplish this task. This section summarizes some of the main features of the X.21 standard. For more detailed information on X.21, see the CCITT X.21 standards document.

In the X.21 protocol the DTE controls two signal lines, Transmit (T) and Control (C). The DTE passes data and signalling information to the DCE over the Transmit line, and passes control information to the DCE on the Control line. The T and C lines are similar in function to the RS-232 TxD and RTS signals, though the T and C lines perform additional duties.

The DCE also controls two signal lines, Receive (R) and Indication (I). The DCE passes user data and signalling information to the DTE on the Receive line, and passes control information to the DTE on the Indication line. The R and I lines are similar in function to the RS-232 RxD and CTS signals, though the R and I lines perform additional duties.

The primary additional duties performed by the T and R lines is the sending and receiving of inband digital signalling information between the DCE and DTE. This signalling information is used in a way that is analogous to the call setup tones you hear when placing a telephone call.

When the line is idle, the DCE always indicates "ready", the equivalent of dial tone. When the DTE initiates a call, the DCE returns a digital "dial tone" indicating that the DTE can continue. The DTE and DCE then exchange a series of call progress messages while they set up a connection with a remote computer. Finally, the DTE and DCE enter a data transfer mode for the duration of the call.

Features of the X.21 Support Hardware

The serial controller performs most X.21 support functions, but there are a few which require special hardware. When the serial controller is in its X.21 mode, it only recognizes data frames which are preceded by two X.21 SYNC characters. This allows it to recognize any valid X.21 data frame, but prevents it from recognizing certain special signalling patterns which can appear in the incoming data bitstream.

Dedicated hardware has been added to detect these special bit patterns. This hardware scans the incoming bitstream for one of four patterns which indicate special conditions in the 1984 X.21 protocol. When it detects one of the patterns, it first verifies that the Indication line is in the correct state for the pattern to be valid. The X.21 hardware then writes a value in a special register and changes the state of the CTS signal to the serial controller.

This causes the serial controller to generate an interrupt whenever the X.21 hardware detects one of the special bit patterns. Note that the X.21 hardware only detects incoming patterns. It does not generate outgoing bit patterns. That is the job of the application program.

Three of these bit patterns must be held by the DCE for at least 24 consecutive bits. They are detected by the X.21 hardware when two consecutive bytes (16 bits) contain the pattern. The fourth pattern must be held by the DCE for 16 bits, and must appear in the data bitstream only once, immediately after the Indication line goes "on." This pattern indicates successful call completion.

See Table 8-2 for a description of each bit pattern, and its required Indication line state.

Table 8-2. X.21 Status Indication Bit Patterns

Bit Pattern	I-Line State	Meaning
00000000 00000000 00000000	Off	DCE Not Ready
11111111 11111111 11111111	Off	DCE Ready
01010101 01010101 01010101	Off	DCE Controlled, not Ready
11111111 11111111	On	Ready For Data (Call Setup Complete)

When the serial controller detects a change in the state of CTS, it generates an External/Status interrupt. The communication program's External/Status ISR must then check that the cause of the interrupt is the X.21 hardware (that the state of CTS has changed) then read the X.21 hardware register. InitCommLine returns the port address of the X.21 hardware register in the *ioX21* field of the Communications Line Return Area.

The possible values of bits 0-2 of the *ioX21* port are as follows. The decimal values are listed, with the corresponding binary values in parentheses.

Value	Description
1 (001b)	DCE Not Ready
2 (010b)	DCE Ready
3 (011b)	DCE Controlled, not Ready
4 (100b)	DCE Ready for Data

The X.21 hardware also has the ability to store one bit pattern in a temporary buffer. Therefore, if a second bit pattern is detected while the first is being processed by the program's ISR, the X.21 hardware stores that pattern until the ISR finishes. It then changes the state of CTS again to generate another interrupt. Note that even if the X.21 hardware has a pattern stored in its buffer, it continues to check the incoming bitstream. If it detects a new pattern, it overwrites the currently-stored one.

Initializing a Communications Line with X.21 Support Enabled

To request use of the X.21 hardware, a communications program must set the *fX21* flag in the Communications Line Configuration Block to the value 0AAh. This value requests full X.21 support.

If the X.21 hardware is present, InitCommLine returns the port address of the X.21 hardware in the *ioX21* field of the Communications Line Return Block. If the hardware is not present, InitCommLine returns a nonzero status code.

After a successful call to InitCommLine, the X.21 hardware for the channel is inactive. To activate the X.21 hardware, the program must write a one to bit zero of the *ioX21* port. After that happens, the X.21 hardware begins scanning the input stream for its special bit patterns.

The program can disable the X.21 hardware at any time, by writing a zero to bit zero of the X.21 hardware port. The program can then reenable it later by writing a one. The program should be careful to write only to bit zero of the X.21 hardware port, however. Other bits are used internally by the Comm Nub.

Note also that programs which use the X.21 hardware must initialize the serial controller to enable the External/Status interrupt. Otherwise, changes in the CTS signal by the X.21 hardware do not cause an interrupt.

Using the X.21 Support Hardware in Drivers-Only Mode

Using the X.21 support hardware in drivers-only mode disables the pattern-recognition circuitry. This allows programs to use the 1981 X.21 protocol, which does not include the bit patterns described above.

To request use of X.21 in drivers-only mode, the communications program must set the *fX21* flag in the Communications Line Configuration Block to the value 0FFh.

If X.21 is supported by the hardware, InitCommLine returns status code 0. However, it does not return a port address in the *ioX21* field of the Communications Line Return Area because the pattern detection hardware is not used in this mode. If X.21 is not supported by the hardware, InitCommLine returns a nonzero status code.

Using V.35 Support Hardware

As with X.21, only certain hardware supports the V.35 interface. However, using this interface is almost a non-event for a communications program. V.35 is an electrical interface, and appears to be the same as RS-232 to communications programs, except that it supports faster data rates.

Existing programs need no modification to use the V.35 support hardware. When InitCommLine receives a Communications Line Configuration Block that is too small to contain the *fV35Mode* flag, it simply uses the current jumper settings for the requested communications port. So, if an existing program attempts to use a port that is jumpered for V.35, the attempt succeeds. Note, however, that such programs still only support the data rates they supported on RS-232. They may or may not work at higher V.35 rates.

New programs should set the *fV35Mode* flag in the Communications Line Configuration Block. This flag informs InitCommLine that the program

wants to use V.35 mode. If the requested communications port supports (and is jumpered for) V.35, InitCommLine sets the *fV35Avail* field in the Communications Line Return Area to TRUE. If the requested communications port does not support (or is not jumpered for) V.35, InitCommLine sets the *fV35Avail* field to FALSE, and returns a nonzero status code.

Programs that explicitly check for V.35 support as described above can use the higher data rates supported by the V.35 interface. Those which do not verify that V.35 hardware is present should restrict themselves to data rates supported by both RS-232 and V.35.

9

The SCSI Manager Target Mode

This chapter describes the target mode of the SCSI Manager. This mode allows applications on a CTOS machine to communicate with other processors that support the SCSI-2 standard.

When the SCSI Manager communicates with peripheral devices such as disks, it uses initiator mode. In initiator mode, the SCSI Manager controls the exchange of data between itself and the peripheral device.

When in target mode, the SCSI Manager reverses its role. Another device on the SCSI bus acts as the initiator, and the SCSI Manager acts as a target device of the *processor* type. This chapter describes the operation of the SCSI Manager as a processor device in target mode.

The operation of the SCSI Manager in initiator mode is described in the *CTOS Operating System Concepts Manual.*

Introduction

In addition to managing SCSI initiator functions to communicate with devices connected to the SCSI bus, the CTOS SCSI Manager can function as a SCSI device of the processor type. This allows it to respond to commands sent by other initiators connected to the SCSI bus.

In target mode, the SCSI Manager can move data efficiently between the CTOS processor and other initiator(s) on the SCSI bus. Given the distance limitations of SCSI, a possible use of this mode might be a Very Local Area Network (VLAN) to connect two computer systems.

SCSI initiators that issue commands to the SCSI Manager must conform to certain minimum levels of SCSI implementation as defined in the SCSI standard, X3.131–1986. These are as follows:

- They must support parity generation. Parity checking by the SCSI initiator is optional but recommended.
- They must conform to level two of the SCSI standard. In particular they must conform to the requirement that Logical Unit Numbers (LUNs) are addressed by means of the IDENTIFY message, not by the LUN field in the Command Descriptor Block.
- They must support the "hard" reset option.

Programmers who use the SCSI Manager target mode to transfer data between SCSI devices should be thoroughly familiar with the SCSI standard definition of processor devices, both their conceptual model and their operation. The section *Guidelines for SCSI Processor Target Mode* in this chapter offers some tips on the use of the SCSI Manager in this mode. The rest of the chapter consists of a detailed specification of the SCSI processor device represented by the SCSI Manager when it is in target mode.

SCSI Commands

The SCSI Manager's target mode functions define a SCSI device in the processor class as specified in the SCSI-2 specification. The processor device implemented by the SCSI Manager is capable of accepting the following SCSI commands:

Value	Command
12h	INQUIRY
08h	RECEIVE
03h	REQUEST SENSE
0Ah	SEND
1Dh	SEND DIAGNOSTIC
00h	TEST UNIT READY

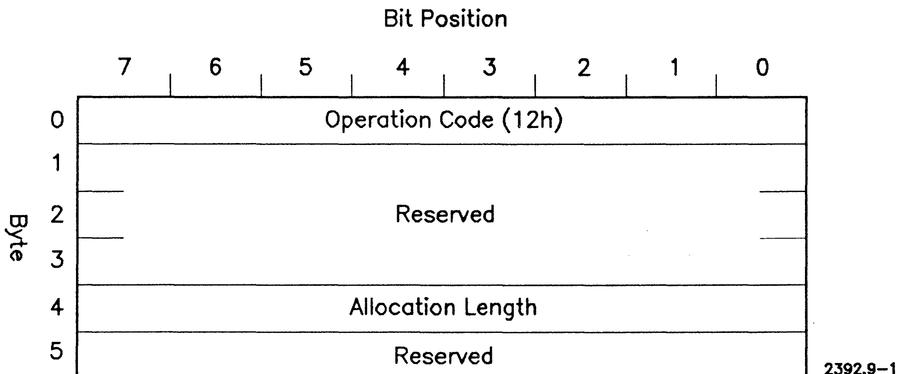
Any other commands received by the SCSI Manager processor device are terminated with CHECK CONDITION status, and the sense key is set to ILLEGAL REQUEST (05h). If no path has been established to the specified LUN, the additional sense code is set to LOGICAL UNIT NOT SUPPORTED (25h). Otherwise, the additional sense code is set to INVALID COMMAND OPERATION CODE (20h).

The INQUIRY, REQUEST SENSE, SEND DIAGNOSTIC and TEST UNIT READY commands are processed internally by the SCSI Manager. They require no intervention on the part of any application program or system service. "Canned" responses are available for these commands even if no program has established a path to the processor device LUN specified. The RECEIVE and SEND commands are rejected by the SCSI Manager unless a program has activated the specified LUN by establishing a path. More detail for each individual command is given below.

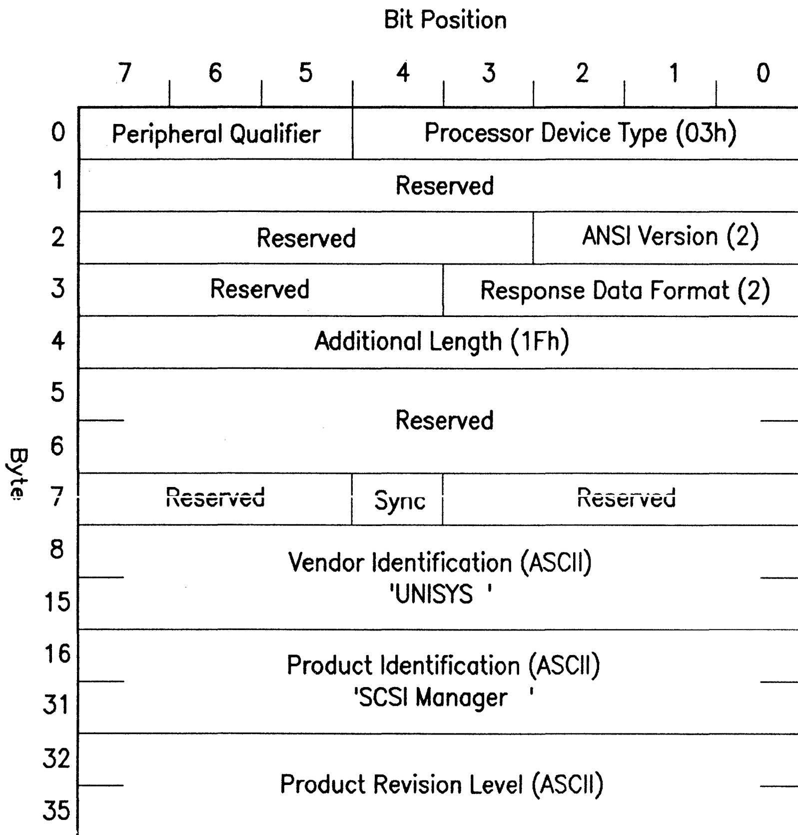
Each command requires a specific format for its Command Descriptor Block (CDB). If the SCSI Manager receives a CDB in which any of the reserved bits are not zero, the command is terminated with CHECK CONDITION status, the sense key is set to ILLEGAL REQUEST (05h) and the additional sense code is set to INVALID FIELD IN CDB (24h).

INQUIRY

The Command Descriptor Block (CDB) for the INQUIRY command accepted by the SCSI Manager is shown below.



The INQUIRY command requests the SCSI Manager to return data that uniquely identifies the SCSI Manager processor device and its capabilities. The SCSI Manager can return up to 36 bytes of inquiry data to the initiator device, as described below.



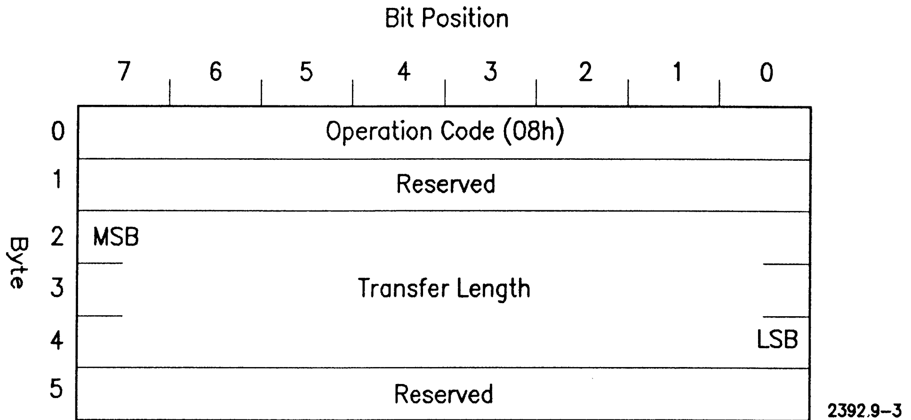
If the SCSI Manager receives an INQUIRY command addressed to a LUN for which no SCSI Manager path has been established, the SCSI Manager sets the Peripheral Qualifier field in byte 0 to a value of 1. This sets byte 0 to a value of 23h, indicating that the desired LUN is not installed.

If the SCSI Manager receives an INQUIRY command addressed to an LUN for which path *has* been established, the SCSI Manager sets the value of the ANSI Version field to 2 and sets the value of the Sync bit to 1. This indicates the presence of a processor device that conforms to the SCSI-2 standard and is capable of synchronous data transfer.

The SCSI Manager either returns all 36 bytes of inquiry data to the initiator, or returns the number of bytes specified in the maximum allocation length field of the INQUIRY command, whichever is less.

RECEIVE

The Command Descriptor Block (CDB) for the RECEIVE command accepted by the SCSI Manager is shown below.



The RECEIVE command requests a transfer of data from the SCSI Manager target mode to a SCSI initiator. The length of the data to be transferred, in bytes, is specified by the Transfer Length field.

If no target-mode path has been opened at the SCSI Manager, the RECEIVE command is terminated with CHECK CONDITION status, the sense key is set to ILLEGAL COMMAND (05h) and the additional sense code is set to LOGICAL UNIT NOT SUPPORTED (25h). If a target mode path for the LUN specified *has* been established by a ScsiOpenPath operation, the RECEIVE command operates in one of two modes, depending upon whether or not a ScsiTargetDataTransmit operation has already been issued for the LUN.

If a buffer is available from which to transmit the data for the RECEIVE command (in other words, a ScsiTargetDataTransmit operation is pending for the LUN), the SCSI Manager transfers the data in the buffer to the SCSI initiator during the DATA IN phase and the RECEIVE command completes normally.

Otherwise, the SCSI Manager disconnects from the SCSI bus after the Command Descriptor Block (CDB) has been received. This CDB may be obtained by an application by means of the ScsiTargetCdbCheck or

ScsiTargetCdbWait operations. A subsequent ScsiTargetDataTransmit operation may provide a buffer with data to transmit, in which case the SCSI Manager reconnects to the selecting initiator, transfers the information during a DATA IN phase and completes the command normally.

If the transfer length of the DATA IN phase does not match the size of the buffer provided by the ScsiTargetDataTransmit operation, the SCSI Manager detects an incorrect length condition and sets the appropriate target status and sense data at the completion of the RECEIVE command.

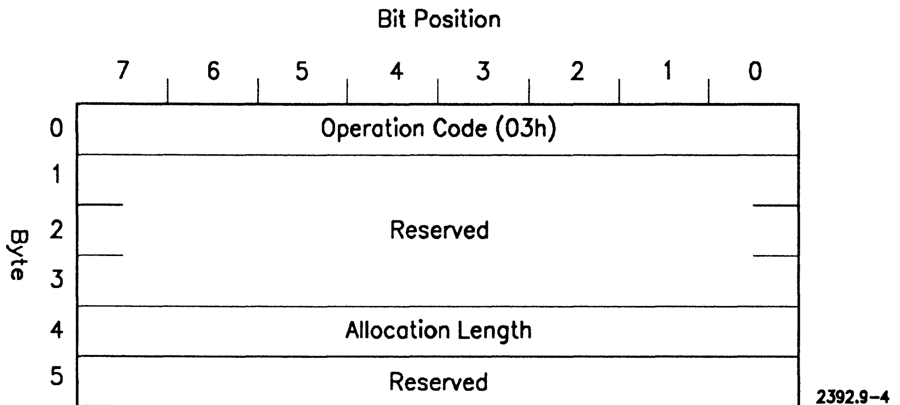
If the Transfer Length is less than the size of the buffer provided, the RECEIVE command completes with GOOD status, the sense key is set to NO SENSE (00h), the Illegal Length Indicator (ILI) in the sense data is set to one and the residue is set to the Transfer Length from the RECEIVE command minus the buffer size. In this case, the residue is a negative number (two's complement notation) since the requested length was smaller than the buffer provided.

This sense data is available to the SCSI initiator that issued the RECEIVE command if that initiator sends REQUEST SENSE as its next command. The same sense data is also returned to the application that issued the ScsiTargetDataTransmit so that it can determine that less data was transferred than requested.

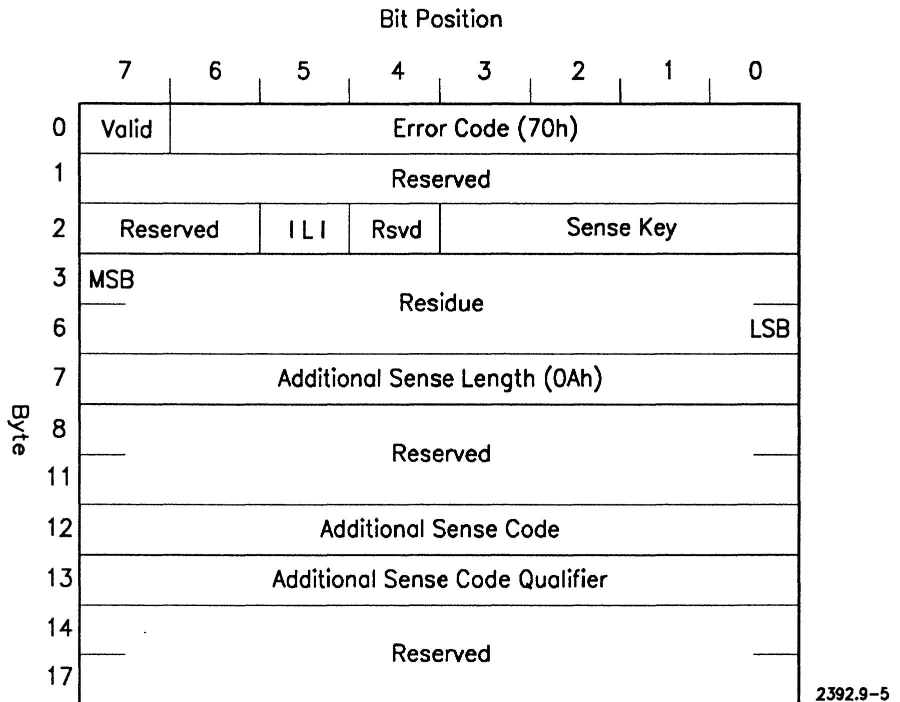
If the Transfer Length is greater than the size of the buffer provided, the RECEIVE command completes with CHECK CONDITION status, the sense key is set to NO SENSE (00h), the Illegal Length Indicator (ILI) in the sense data is set to one and the residue is set to the Transfer Length from the RECEIVE command minus the buffer size. In this case, the residue is a positive number since the requested length was larger than the buffer provided. The target status and sense data are also returned to the application that issued the ScsiTargetDataTransmit so that it can determine that an attempt was made to transfer more data than requested.

REQUEST SENSE

The command descriptor block (CDB) for the REQUEST SENSE command accepted by the SCSI Manager is shown below. The Allocation Length field specifies how many bytes of sense data the SCSI Manager should return, up to a maximum of 18 bytes.



When the SCSI Manager receives the REQUEST SENSE command, it returns any available sense data to the initiator, then clears the data. Sense data is created at the end of each command, and in response to a UNIT ATTENTION condition. The sense data format is shown below.



A value of zero in the Valid bit indicates that the contents of the residue field are not defined. A value of one in the Valid bit indicates that the contents of the residue field reflects the difference between the requested transfer length in a RECEIVE or SEND command and the quantity of data actually transferred. Negative values are indicated by two's complement notation.

Error code 70h indicates a current error. The SCSI Manager target mode does not support deferred errors.

An Illegal Length Indicator (ILI) bit of one indicates that the transfer length requested by the initiator did not match the size of the buffer area provided to the SCSI Manager by a ScsiTargetDataReceive or ScsiTargetDataTransmit operation. The information in the residue field is valid and may be used to determine the extent of the overrun or underrun.

The sense key field indicates the generic nature of the error or exception condition. The SCSI Manager reports only the sense keys listed below.

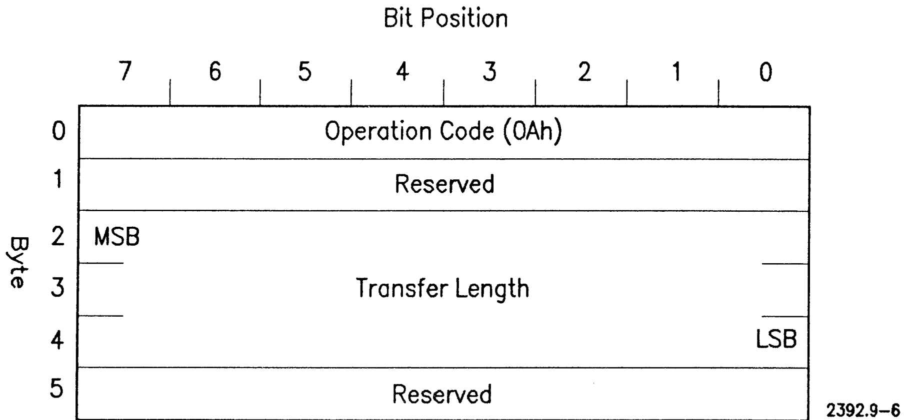
Value	**Sense Key**
00h	NO SENSE
05h	ILLEGAL REQUEST
06h	UNIT ATTENTION
0Bh	ABORTED COMMAND

The additional sense code field contains further information related to the error or exception condition reported in the sense key field. The additional sense codes reported by the SCSI Manager are listed below.

Value	**Additional Sense Code**
00h	NO ADDITIONAL SENSE INFORMATION
20h	INVALID COMMAND OPERATION CODE
24h	INVALID FIELD IN CDB
25h	LOGICAL UNIT NOT SUPPORTED
29h	POWER ON, RESET OR BUS DEVICE RESET OCCURRED
3Dh	INVALID BITS IN IDENTIFY MESSAGE
47h	SCSI PARITY ERROR
48h	INITIATOR DETECTED ERROR MESSAGE RECEIVED
4Eh	OVERLAPPED COMMANDS ATTEMPTED

SEND

The Command Descriptor Block (CDB) for the SEND command accepted by the SCSI Manager is shown below.



The SEND command requests a data transfer from a SCSI initiator to the SCSI Manager in target mode. The length of data to be transferred, in bytes, is specified by the Transfer Length field.

If no path has been opened in target mode at the SCSI Manager, the SCSI Manager terminates the SEND command with CHECK CONDITION status, sets the sense key to ILLEGAL COMMAND (05h) and sets the additional sense code to LOGICAL UNIT NOT SUPPORTED (25h). If a target mode path for the LUN specified has already been established by a ScsiOpenPath operation, the SEND command operates in one of two modes, depending upon whether or not a ScsiTargetDataReceive operation has already been issued for the LUN.

If a buffer is available to receive the data from the send command (in other words, a ScsiTargetDataReceive operation is pending for the LUN), the SCSI Manager transfers any data obtained during the DATA OUT phase to the buffer specified in the ScsiTargetDataReceive operation and the SEND command terminates normally.

Otherwise, the SCSI Manager disconnects from the SCSI bus after the Command Descriptor Block (CDB) has been received. This CDB may be obtained by an application by means of the ScsiTargetCdbCheck or

ScsiTargetCdbWait operations. A subsequent ScsiTargetDataReceive operation should provide a buffer for the data, in which case the SCSI Manager reconnects to the selecting initiator, transfers the information during a DATA OUT phase and completes the command normally.

If the transfer length of the DATA OUT phase does not match the size of the buffer provided by the ScsiTargetDataReceive operation, the SCSI Manager detects an incorrect length condition and sets the appropriate target status and sense data at the completion of the SEND command.

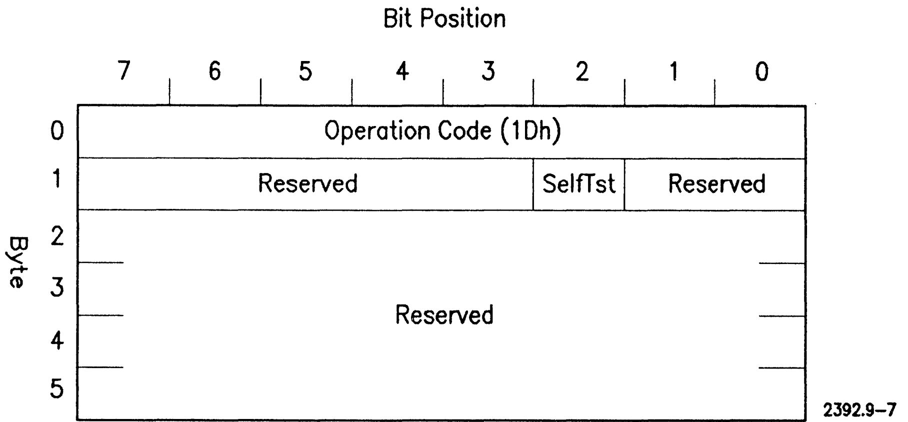
If the Transfer Length is less than the size of the buffer provided, the SCSI Manager completes the SEND command with GOOD status, sets the sense key to NO SENSE (00h), and sets the Illegal Length Indicator (ILI) in the sense data. It also sets the residue to the Transfer Length from the SEND command minus the buffer size. In this case, the residue is a negative number (two's complement notation) since the requested length was smaller than the buffer provided.

This sense data is available to the SCSI initiator that issued the SEND command if that initiator sends REQUEST SENSE as its next command. The SCSI Manager also returns the same sense data to the application that issued the ScsiTargetDataReceive so that it can determine that less data was transferred than requested.

If the Transfer Length is greater than the size of the buffer provided, the SCSI Manager completes the SEND command with CHECK CONDITION status, sets the sense key to NO SENSE (00h), and sets the Illegal Length Indicator (ILI) in the sense data. It also sets the residue to the Transfer Length from the SEND command minus the buffer size. In this case, the residue is a positive number since the requested length was larger than the buffer provided. The SCSI Manager also returns target status and sense data to the application that issued the ScsiTargetDataReceive, so that it can determine that an attempt was made to transfer more data than requested.

SEND DIAGNOSTIC

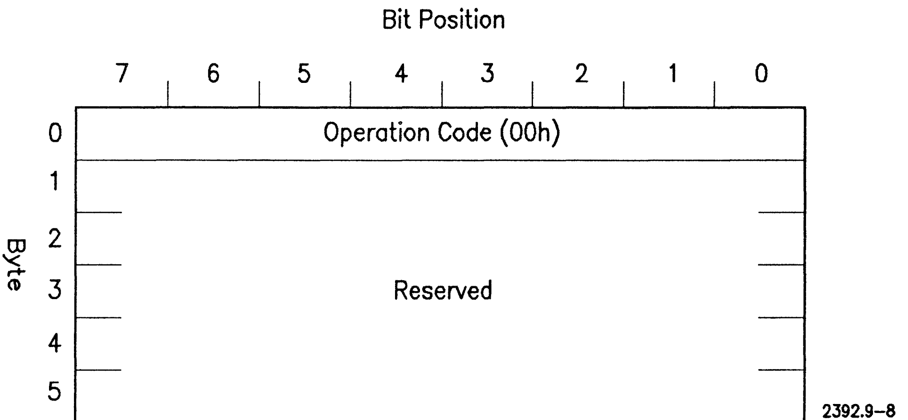
The command descriptor block (CDB) for the SEND DIAGNOSTIC command accepted by the SCSI Manager is shown below.



The SEND DIAGNOSTIC command requests the SCSI Manager target mode to perform self-test diagnostic functions. If there is no pending UNIT ATTENTION condition, GOOD status is returned by the command; otherwise SEND DIAGNOSTIC terminates with a CHECK CONDITION status.

TEST UNIT READY

The command descriptor block (CDB) for the TEST UNIT READY command accepted by the SCSI Manager is shown below.



TEST UNIT READY may be used by another SCSI initiator to verify that the SCSI Manager is ready to receive or transmit data in target mode. If a SCSI Manager path in target mode has been established for the LUN specified by the TEST UNIT READY command, the command completes with GOOD status. Otherwise, TEST UNIT READY terminates with CHECK CONDITION status, the sense key is set to ILLEGAL REQUEST (05h) and the additional sense code is set to LOGICAL UNIT NOT SUPPORTED (25h).

SCSI Messages

The SCSI Manager processor target mode accepts the following messages from an initiator during a MESSAGE OUT phase. Any other commands are rejected with a MESSAGE REJECT message.

Value	Message
06h	ABORT
0Ch	BUS DEVICE RESET
80h	IDENTIFY
05h	INITIATOR DETECTED ERROR
09h	MESSAGE PARITY ERROR
07h	MESSAGE REJECT
08h	NO OPERATION
***	SYNCHRONOUS DATA TRANSFER REQUEST

The SCSI Manager processor target mode generates the following messages during a MESSAGE IN phase.

Value	Message
00h	COMMAND COMPLETE
04h	DISCONNECT
80h	IDENTIFY
07h	MESSAGE REJECT
***	SYNCHRONOUS DATA TRANSFER REQUEST

For those messages that require additional description of the actions taken by the SCSI Manager target mode (beyond the specifications in the SCSI-2 standard), more information is given below.

ABORT

The ABORT message is sent from an initiator to clear the current I/O process for the specified LUN. Any pending data and status for the identified LUN and the issuing initiator are cleared and the SCSI Manager target mode enters the BUS FREE phase. If no LUN has been identified when an ABORT message is received, there is no pending data or status to clear, so the SCSI Manager enters the BUS FREE phase.

If there are any pending ScsiTargetCdbWait, ScsiTargetDataReceive or ScsiTargetDataTransmit operations on the specified LUN for the initiator that sent the ABORT message (or any that were issued with the initiator ID parameter set to "don't care"), they are terminated and error code 379 ("SCSI command aborted") is returned to the program that issued them.

BUS DEVICE RESET

The BUS DEVICE RESET message is sent from an initiator to clear all I/O processes for the SCSI Manager target mode. All pending data and status for the issuing initiator are cleared and the SCSI Manager target mode enters the BUS FREE phase. A unit attention condition is created for all initiators.

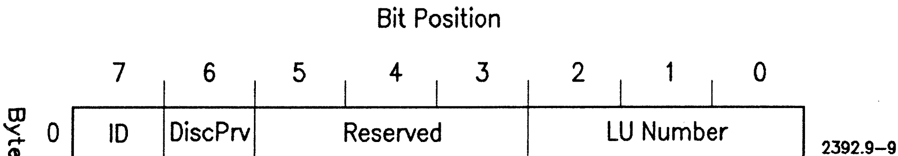
Any pending ScsiTargetCdbWait, ScsiTargetDataReceive or ScsiTargetDataTransmit operations are terminated and error code 386 ("SCSI bus reset") is returned.

DISCONNECT

This message is sent by the SCSI Manager target mode when it has received a RECEIVE or SEND command but a matching ScsiTargetDataTransmit or ScsiTargetDataReceive operation has not been issued for the specified LUN. When one of these operations is issued (i.e. a buffer is provided for the transmission or reception of data associated with the RECEIVE or SEND command), the SCSI Manager reconnects to the initiator to complete the I/O process.

IDENTIFY

The IDENTIFY message is sent either by an initiator or by the SCSI Manager to establish a connection between an I/O process of the SCSI Manager target mode and the initiator.



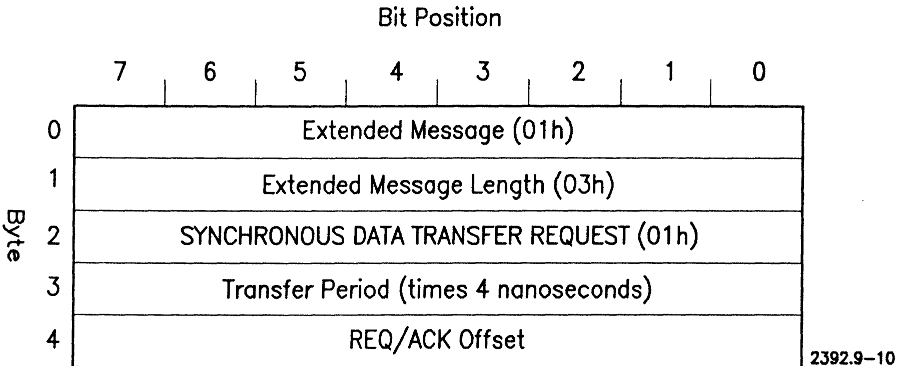
With the exception of the RECEIVE and SEND commands, the SCSI Manager target mode does not disconnect from the SCSI bus until the completion of an I/O process.

If a RECEIVE or SEND command is preceded by an IDENTIFY message that disallows disconnection privileges for the SCSI Manager target mode (*i.e.* the DiscPrv bit is zero) the SCSI Manager cannot suspend that process. If the SCSI Manager encounters a condition in which it would normally disconnect and suspend the I/O process (see the description of the DISCONNECT message above), it terminates the command with CHECK CONDITION status. It also sets the sense key to ABORTED COMMAND and sets the additional sense code to INVALID BITS IN IDENTIFY MESSAGE.

INITIATOR DETECTED ERROR

The INITIATOR DETECTED ERROR message is sent from an initiator to inform the SCSI Manager target mode that an error has been detected during the current I/O process. When it receives an INITIATOR DETECTED ERROR message, the SCSI Manager terminates the current command with CHECK CONDITION status. It then sets the sense key to ABORTED COMMAND and sets the additional sense code to INITIATOR DETECTED ERROR MESSAGE RECEIVED.

SYNCHRONOUS DATA TRANSFER REQUEST



If the SCSI Manager is not capable of synchronous data transfers because of hardware limitations, it always refuses incoming synchronous data transfer requests. If an initiator requests synchronous data transfer, the SCSI Manager returns a SYNCHRONOUS DATA TRANSFER REQUEST message with the REQ/ACK offset set to zero to indicate asynchronous transfer.

If the SCSI Manager is capable of synchronous data transfers, it always accepts SYNCHRONOUS DATA TRANSFER REQUEST messages and attempts to negotiate the shortest transfer period and the greatest REQ/ACK offset acceptable to both the initiator and the SCSI Manager target mode. Unless specifically disabled in the configuration file, the SCSI Manager target mode initiates synchronous data transfer negotiations on the first connection with an initiator if any previously negotiated agreement has become invalid.

The SCSI Manager target mode and an initiator lack a valid synchronous data transfer agreement under any of the following conditions:

- after the SCSI Manager is first initialized
- after a SCSI bus reset
- after receipt of a BUS DEVICE RESET message

Guidelines for SCSI Processor Target Mode

The effective use of SCSI processor target mode to exchange information with another initiator on the SCSI bus (usually another computer system) requires coordination with the application executing on the other system. Items which need to be defined include layouts of message packets, maximum length considerations for message packets, flow control, which party (or both) is responsible for error recovery, and so on. Most of these considerations are beyond the scope of this chapter, but some important aspects of SCSI Manager behavior can be described here. These aspects should be considered in the system design.

There are two fundamentally different ways in which the RECEIVE and SEND commands can be used to transfer information. In the first, data is to be exchanged with only one other initiator on the SCSI bus and the length (or at least, the maximum length) of each transfer is known beforehand. This might be described as *single-threaded* use of the processor target mode. In the second, neither the origin of the data (which initiator sent it) nor the length of the data is known beforehand. This might be called a *multi-threaded* use of processor target mode. Brief examples of both modes are described below.

Single-Threaded Mode

In this mode, the application on the CTOS processor first issues a ScsiTargetDataReceive operation to await a message packet sent from the other initiator by means of a SEND command. The maximum length of any message packet must be known beforehand or the operation may terminate with target status of CHECK CONDITION and an Illegal Length Indicator in the sense data. This ScsiTargetDataReceive operation remains pending until a SEND command is received; there is no time-out for target mode operations.

When the SCSI Manager has received a SEND command and transferred the associated data to the application's buffer, the application may examine the Command Descriptor Block (CDB) and the data. The returned data fields that must be checked are the error return code (it should be zero), the target status (it should be GOOD) and the length of data transferred (it should match the Transfer Length in the SEND CDB). If any of these conditions are not met, an exception condition has occurred and the program should transfer control to its error recovery

procedures. Otherwise, the data received by the application indicates an action to be performed and (usually) a response to be sent to the initiator.

The application responds to the initiator by means of the ScsiTargetDataTransmit operation. When the initiator issues a RECEIVE command, it receives the data from the buffer provided by the ScsiTargetDataTransmit operation and the circle of inbound and outbound message packets is complete. The application can now resume its wait for a message packet from the initiator.

Note that this single-threaded example implements a one-way control of the information exchanged over the SCSI bus: the SCSI initiator is in control of the data transfers and is the one that issues all SCSI commands. The SCSI bus may be used in a duplex, but still single-threaded, fashion if the application on the CTOS processor issues its own SEND and RECEIVE commands to the other SCSI initiator. These commands must be issued by the initiator mode SCSI manager operations in the same fashion as commands to any SCSI device (such as a disk or a tape drive).

Multi-Threaded Mode

This mode allows an application to establish multiple communications sessions to exchange data with more than one other initiator on the SCSI bus. The application awaits (or periodically checks for) the arrival of a SEND command from another initiator by means of the ScsiTargetCdbWait or the ScsiTargetCdbCheck operation. When a SEND command arrives, the SCSI Manager returns the ID of the initiator that issued the command and returns the SEND CDB (which contains the length of data to transfer) to the application. The SCSI Manager disconnects from the SCSI bus after receiving the CDB and is free to continue with other operations. The application can then use the ScsiTargetDataReceive operation to cause the SCSI Manager to reconnect to the initiator and to transfer the data from the initiator, completing the SEND command.

Then, the application can prepare the response which is implicitly requested by the SEND command and queue it for later transmission to the initiator. In this case, the application does not perform a ScsiTargetDataTransmit at this time. Instead, the application updates its own data structures to reflect the fact that a buffer is ready to be sent in

response to a RECEIVE command from a particular initiator if such a command ever arrives. Upon the arrival of a RECEIVE command (detected by the successful completion of a ScsiTargetCdbWait or ScsiTargetCdbCheck operation), the application performs a ScsiTargetDataTransmit operation to cause the response data to be transferred back to the initiator.

Just as in the single-threaded case, a multi-threaded application may use the SCSI bus in a duplex fashion by issuing its own SEND and RECEIVE commands to other SCSI devices with which it wishes to communicate.

Illegal Transfer Lengths

Illegal length (overflow or under-run) conditions occur when the Transfer Length field of a RECEIVE or SEND command is not exactly equal to the size of the buffer provided by a ScsiTargetDataTransmit or ScsiTargetDataReceive operation. In both cases, the sense data associated with the command is updated so the sense key is NO SENSE (00h), and the Illegal Length Indicator is set to one. In addition, the Valid bit is set to one, indicating that the Residue field contains the difference between the expected and actual length of data transferred. Finally, the additional sense code is set to NO ADDITIONAL SENSE DATA (00h). Beyond this point, overflow and under-run are handled differently.

Under-run occurs when the buffer provided to the SCSI Manager is larger than the data sent from or received by the other initiator. In this case, both the RECEIVE and SEND commands complete with GOOD status. The initiator is never made aware of the under-run condition unless it issues a REQUEST SENSE command immediately after the command that caused the under-run. The application using the SCSI Manager by means of a ScsiTargetDataTransmit or ScsiTargetDataReceive operation can detect the under-run condition by examining the sense data for the information described above. Under-run is generally harmless and does not require any error handling.

Overflow occurs when the buffer provided to the SCSI Manager is smaller than the data sent from or received by the other initiator. In this case, both the RECEIVE and SEND commands complete with CHECK CONDITION status. The initiator is normally expected to discover the overflow condition by issuing a REQUEST SENSE command immediately

after the command that caused the overflow. The application using the SCSI Manager by means of a ScsiTargetDataTransmit or ScsiTargetDataReceive operation can detect the overflow condition by examining the target status for CHECK CONDITION and the sense data for the information described above. Overflow usually indicates a programming error, that is, a failure of agreement between the applications executing on the CTOS processor and the other processor.

Finishing a Target Mode Application

When an application using SCSI Manager processor target mode is complete, it needs to close the target mode paths before it exits. Normally, the application would simply call ScsiClosePath for each of the paths it had opened.

However, the ScsiTargetDataReceive, ScsiTargetDataTransmit and ScsiTargetCdbWait operations have no time limit for completion, since they wait for commands from a remote SCSI initiator. If the application attempts to close a SCSI path on which any of these operations are outstanding, the attempt fails with status code 394 ("SCSI requests outstanding"). In this case, the application must call ScsiTargetOperationsAbort for that SCSI path, to cancel any outstanding operations.

When the application calls ScsiTargetOperationsAbort for a path, the SCSI Manager cancels any pending operations on that path unless it has received a Command Descriptor Block (CDB) for one of them. When the SCSI Manager cancels outstanding operations, it returns status code 379 ("SCSI Command Aborted") to the program or process which called each outstanding operation. If it was able to abort all the outstanding operations for a path, the SCSI Manager returns status code zero to the program or process that called ScsiTargetOperationsAbort, indicating that the path can now be closed by a ScsiClosePath operation.

If the SCSI Manager received a CDB for any of the outstanding requests, ScsiTargetOperationsAbort returns error code 394 ("SCSI requests outstanding") and the SCSI path may not be closed. In this case, the application should complete the outstanding operations by normal means. It should then examine the data received to determine an appropriate action before it closes its SCSI paths and exits. For example, if the SCSI Manager received a SEND command CDB, the application should issue a

ScsiTargetDataReceive operation before closing the path. The application should also reply to the SCSI initiator that sent the CDB before exiting.

10

Making CTOS Requests from MS-DOS

Why Call CTOS from DOS?

Many installations include both CTOS workstations and PCs. DOS-based PCs can be integrated into the CTOS cluster using the ClusterCard with ClusterShare software. These products allow PCs to access files and printers at the CTOS server.

They also create the potential for truly integrated applications that run on both the DOS and CTOS platforms. Two examples of this type of integrated application are ClusterShare 3270 and ClusterShare Mail. These applications extend CTOS SNA network access and access to the CTOS mail system to PC users.

This chapter describes how to create similar distributed applications. This type of application allows the user interface to reside on either PCs or CTOS workstations, while the underlying services reside on a CTOS server.

What Is CSKNAMES.OBJ?

A file called CSKNAMES.OBJ is distributed with the ClusterShare software. This file is an object module which allows DOS programs to call CTOS kernel primitives at the server.

This allows a program on a PC to access request-based system services on a CTOS server. The interface from DOS to CTOS is basically identical to the CTOS request interface, though less rich in features.

Kernel Primitives Supported by CSKNAMES.OBJ

CSKNAMES.OBJ allows DOS programs to call several of the CTOS primitives. The syntax used to call them is identical in both DOS and CTOS. CSKNAMES.OBJ supports the following CTOS kernel primitives:

- AllocExch

  DOS programs use AllocExch to allocate an exchange for use in communicating with the server. A DOS program can allocate up to 20 exchanges.

- Check

  Check allows a DOS application to poll an exchange for a response from the server.

- DeAllocExch

  DeAllocExch frees a previously allocated exchange. DOS programs must make sure that they always call DeAllocExch for each exchange they allocate. ClusterShare exchanges are not freed unless DeAllocExch is explicitly called.

- Request

  Request allows a DOS application to send a request to the server. All requests are routed to the server automatically. There is no support for system services under DOS.

- Wait

  Wait causes the DOS application to block until it receives a response to a previous Request. Internally, Wait performs a busy-loop, repeatedly polling a single exchange for a response from the server. This can prevent any other application program from running on the DOS machine until a response is received.

The main difference in the handling of kernel primitives under DOS is that an exchange can have only one outstanding request. If a program makes only one Request at a time then Waits for a response to it, this fact is insignificant. However, if a program may have multiple requests outstanding, it must use a separate exchange for each of them.

This fact impacts the way such programs process responses to their requests, as well. Under CTOS, the program could simply Wait at a single exchange and receive each response in turn. Under DOS, however, the program must loop, using Check to poll each exchange for a response.

Using CSKNAMES.OBJ

Using CSKNAMES.OBJ is much like using the kernel primitives under CTOS. For detailed information on using CTOS kernel primitives, see the *CTOS/Open Programming Practices and Standards* guide. There are a few special concerns in the DOS environment, though.

First, you need to define the kernel primitives as external functions, and you need to link CSKNAMES.OBJ with your program. While most CTOS languages provide an external definition file, you have to do this yourself in the DOS environment. Listing 10-1 shows sample C-language function definitions for each of the kernel primitives.

```
extern int    pascal AllocExch(int far *exch);
extern int    pascal Check(int exch, void far *MsgRet);
extern int    pascal DeAllocExch(int exch);
extern int    pascal Request(void far *RqBlk);
extern int    pascal Wait(int exch, void far *MsgRet);
```

Listing 10-1. DOS Function Definitions for the Kernel Primitives

Note that the kernel primitives use the same calling convention as CTOS. This is compatible with the Pascal calling convention in most DOS compiler implementations. See your compiler documentation for more specific information on mixed-language programming.

Second, you need to link CSKNAMES.OBJ with your program. The CTOS Linker automatically links the needed files for you, but DOS linkers do not. Check your DOS compiler or linker documentation for information on linking multiple object modules.

Third, you must use a model of computation which uses far procedural calls (as opposed to near calls). These models include Medium, Large, and Huge. Also, when you pass a pointer as a parameter to one of the kernel primitives, that pointer must be far.

Finally, as mentioned above, each DOS exchange can have only one outstanding request at a time. If your program may have multiple requests outstanding at one time, you must use a separate exchange for each of them.

If you follow these guidelines, your DOS program should be able to communicate with CTOS successfully. See the sample program at the end of this chapter for an example.

The PC Emulator Version Port

DOS programs which run under the PC Emulator can read I/O port 7CE0h to determine which version of the PC Emulator is present. DOS programs can also use this function to determine whether they are running under the PC Emulator or on an actual PC.

Listing 10-2 shows an Assembly language procedure, EmuVersion, which reads the PC Emulator version number. Listing 10-3 shows a C procedure which calls EmuVersion. If the PC Emulator is present, a nonzero byte value will be returned, which represents the major and minor version numbers of the PC Emulator. The high-order four bits of the byte contain the major version number, and the low-order four bits contain the minor version number.

For example, if the PC Emulator Version 4.0 is present, the byte value returned is 40h.

```
PUBLIC _EmuVersion
_EmuVersion PROC NEAR
   ;Returns PCEmulator version in AL
   Push bp
   Mov  bp,sp
   Mov  AX,0
   Mov  DX,7CE0h
   In   AL,DX
   Pop  bp
   Ret
_EmuVersion ENDP
```

Listing 10-2. Determining PC Emulator Version From DOS

```
VersionCheck()  {
   Version = EmuVersion();
   if(Version <= 0)
      /* on a pc */
      DoPCStuff();
   else if(Version < 0x30) {
      /* older than 3.0 */
      printf("\nBuy new software!");
      exit(0);
      }
   else
      /* reasonably up to date emulator version */
      DoEmulatorStuff();
}
```

Listing 10-3. Calling a PC Emulator Version Procedure

A Sample Program Using CSKNAMES.OBJ

The following sample DOS program retrieves the current time from the CTOS server and displays it. It then allows the DOS user to change the time at the CTOS server.

This program has been observed to work correctly with Turbo C 2.0 and with Microsoft C 5.1.

Note that the program must be linked with CSKNAMES.OBJ.

```
/****************************************
 * Filename:      Time.c
 * Author:        A. Coleman/K. Fuiks
 * Date:          12/12/89
 * Compiler:      Turbo C 2.0
 *
 * This file changes the CTOS time from the DOS environment.  This
 * program can be used from a PC with ClusterCard or from the PC
 * Emulator.  ClusterShare must be installed in either case.
 *
 * This program is currently written for Medium model (far code,
 * near data), but can be compiled as Large.  CSKNAMES.OBJ cannot be
 * used with near code models, such as Small and Compact.
 ****************************************/

#include <stdio.h>
#include "process.h"
#include "ctype.h"

typedef struct {        /* request block header */
   char   sCntInfo;
   char   RtCode;
   char   nReqPbCb;
   char   nRespPbCb;
   int    userNum;
   int    exchResp;
   int    ercRet;
   int    rqCode;
   }   RqBlkHdr;

typedef struct {      /* pbcbs */
   void far *pb;
   int       cb;
   }   pbcb;

typedef struct {             /* define a full request block */
   RqBlkHdr   Header;
   pbcb       Items[2];
   }   RequestBlock1;

typedef  struct {            /* define a full request block */
   RqBlkHdr       Header2;
   unsigned int   seconds;
   unsigned int   dayTimes2;
   }   RequestBlock2;
```

Listing 10-4. A Sample DOS Program Using CSKNAMES.OBJ (Page 1 of 5)

```
RequestBlock1      GetTimeReq;          /* create one */
RequestBlock1 far *GetTimePtr = &GetTimeReq;  /* and a pointer to it */
RequestBlock2      SetTimeReq;        /* create one */
RequestBlock2 far *SetTimePtr = &SetTimeReq; /* and a pointer to it */

unsigned int    seconds;
unsigned int    dayTimes2;

int       Exchange = 0;
int far *Exchpoint = &Exchange;
int       erc = 0;

extern int       pascal AllocExch(int far *exch);
extern int       pascal Check(int exch, void far *MsgRet);
extern int       pascal DeAllocExch(int exch);
extern int       pascal Request(void far *RqBlk);
extern int       pascal Wait(int exch, void far *MsgRet);
int       BuildReqBlk1();
int       BuildReqBlk2();
int       GetDateTime();
int       PrintCTOSDateTime();
int       GetNewDateTime();
int       SetNewDateTime();
int       CheckErc(int x);

typedef struct {    /* this is the CTOS date/time structure */
   unsigned   int   seconds;
   unsigned   int   dayTimes2;
   }   DateTimeStruc;

DateTimeStruc      CTOSTime;
DateTimeStruc far *TimeRet = &CTOSTime;

main()
{
int erc;

   erc = AllocExch(Exchpoint);
   CheckErc( erc);
   CheckErc(GetDateTime());
   CheckErc( GetNewDateTime());
   if(seconds) CheckErc( SetNewDateTime());
   PrintCTOSDateTime();
   DeAllocExch(Exchange);
}
```

Listing 10-4. A Sample DOS Program Using CSKNAMES.OBJ (Page 2 of 5)

```
GetDateTime()
{
   int erc;
   BuildReqBlkl();                /* build request block */
   if(Exchange)
   {
      CheckErc( Request(GetTimePtr));
      printf("Sent Get Time request.\n");
      erc = Wait(Exchange, &GetTimePtr);
      CheckErc(erc);
   }

   PrintCTOSDateTime();
   return(0);
}

BuildReqBlkl()
{
   GetTimeReq.Header.sCntInfo = 6;
   GetTimeReq.Header.RtCode = 0;
   GetTimeReq.Header.nReqPbCb = 0;
   GetTimeReq.Header.nRespPbCb = 1;
   GetTimeReq.Header.userNum = 0;
   GetTimeReq.Header.exchResp = Exchange;
   GetTimeReq.Header.rqCode = 14;
   GetTimeReq.Items[1].pb = TimeRet;
   GetTimeReq.Items[1].cb = 4;
}

PrintCTOSDateTime()
{
unsigned   int totalsecs = 0;
int         iHour;
char sec[] = "00", min[] = "00", hour[] = "00", ampm[2];

/* --- Figure out total seconds since last noon or midnight --- */
   totalsecs = CTOSTime.seconds;
/* --- Figure out seconds --- */
   if ((sprintf(sec, "%d", (totalsecs % 60))) < 2)
   {
      sec[1] = sec[0];
      sec[0] = '0';
   }
```

Listing 10-4. A Sample DOS Program Using CSKNAMES.OBJ (Page 3 of 5)

```
/* ---- Figure out minutes ----- */
   if ((sprintf(min, "%d", ((totalsecs / 60) % 60))) < 2)
   {
      min[1] = min[0];
      min[0] = '0';
   }
/* ----- Figure out hours and AM/PM ---- */
   iHour = (totalsecs/(60*60));
   if(iHour == 0)
      iHour = 12;
   sprintf( hour, "%d", iHour);
   if( !(CTOSTime.dayTimes2 % 2))
      sprintf( ampm, "AM");
   else
      sprintf( ampm, "PM");
/* ------- print it ------- */
   printf("The CTOS time is:  %s:%s:%s %s \n\r", hour, min, sec, ampm);
}

GetNewDateTime()
{
char   ampm[10];
unsigned int hours=0, mins = 0, secs = 0;
   printf("Enter new CTOS Time, or any letter to exit: ");
   scanf("%d:%d:%d %2s", &hours, &mins, &secs, ampm);
   strupr(ampm);
   switch (ampm[0]) {
   case 'P':
      {
      if(CTOSTime.dayTimes2 % 2)
         dayTimes2 = CTOSTime.dayTimes2;
      else
         dayTimes2 = CTOSTime.dayTimes2 + 1;
      break;
      }
   case 'A':
      {
      if(!(CTOSTime.dayTimes2 % 2))
         dayTimes2 = CTOSTime.dayTimes2;
      else
         dayTimes2 = CTOSTime.dayTimes2 - 1;
      break;
      }
   default: return(1);
   } /* end of case */
```

Listing 10-4. A Sample DOS Program Using CSKNAMES.OBJ (Page 4 of 5)

```
    if(secs < 60 && mins < 60 && hours <= 12 && hours >= 1) {
        seconds = secs + mins * 60 + hours * 60 * 60;
        return(0);
    }
    else return(1);
}

BuildReqBlk2() {
    SetTimeReq.Header2.sCntInfo = 4;
    SetTimeReq.Header2.RtCode = 0;
    SetTimeReq.Header2.nReqPbCb = 0;
    SetTimeReq.Header2.nRespPbCb = 0;
    SetTimeReq.Header2.userNum = 0;
    SetTimeReq.Header2.exchResp = Exchange;
    SetTimeReq.Header2.rqCode = 51;
    SetTimeReq.seconds = seconds;
    SetTimeReq.dayTimes2 = dayTimes2;
}

SetNewDateTime() {
    BuildReqBlk2();
    if(Exchange) {
        CheckErc( Request(SetTimePtr));
        printf("I sent the Set Time request.\n");
        CheckErc(Wait(Exchange, &SetTimePtr));
        printf("I sent the Get Time request.\n");
        CheckErc( Request(GetTimePtr));
        CheckErc(Wait(Exchange, &GetTimePtr));
        }
    return(0);
}

CheckErc(int error) {
    int erc;
    if(error) {
        printf("An error has occurred.  erc = %d\n\r", error);
        if(Exchange)
            erc = DeAllocExch(Exchange);
        exit(erc);
    }
    else
        return(0);
}
```

Listing 10-4. A Sample DOS Program Using CSKNAMES.OBJ (Page 5 of 5)

Index

82530 serial controller, I:8-3
  EOF reporting, I:8-7
8274 serial controller, I:8-3

a-characters, CD-ROM files, II:7-56
a1-characters, CD-ROM files, II:7-57
Abort requests, II:6-17, 6-19
Accessing a remote queue, II:2-22
Adaptive Differential Pulse Code Modulation (ADPCM), II:4-18
AddQueue operation, II:2-1, 2-7, 2-14, 2-17, 2-21
AddQueueEntry operation, II:2-3, 2-7, 2-12, 2-21, 3-7
Address space protection, I:2-12
Aliasing, I:2-9
Allocating heap memory, II:6-15
AllocMemoryInit procedure, II:6-32
AlphaColorEnabled, I:3-24, 3-25
Alt requests, I:4-3
Amplifying voice messages, II:4-16
Analog crosspoint switch array, II:4-11
Analog-to-digital signal conversion, II:4-2
Application programs
  as queue servers, II:2-5
Applications
  using the spooler, II:3-3
AsGetVolume, II:4-7, 4-13
AsSetVolume, II:4-7, 4-13, 4-16
Async.lib procedures
  in common-code module, II:6-25 to 6-33
  in main module, II:6-10 to 6-24
Async.lib, II:6-1
Asynchronous model
  advantages of, II:6-3
  example of, II:6-4
Asynchronous processing, II:6-2
  diagram of, II:6-7

Asynchronous request procedural interface, II:6-10
Asynchronous system service model, II:6-1
Asynchronous Terminal Emulator (ATE), II:4-3
AsyncRequest procedure, II:6-14
AsyncRequestDirect procedure, II:6-14
AsyncStats, II:6-23
Attribute byte, I:3-22
At-files, II:6-34
Audio features, of CD-ROM, II:7-21
Audio management, II:4-4
Audio Pause, II:7-22
Audio play function, of CD-ROM Service, II:7-21
Audio Play, example of, II:7-22
Audio Q-Channel Info function, of CD-ROM Service, II:7-22
Audio Resume, II:7-22
Audio Service, II:4-1, 4-2
Audio Status, II:7-22
Awk, I:1-2

B25/NGEN workstations, II:4-1
Background color, I:3-2
Backslash, used in CD-ROM file specification, II:7-7
Banner page, II:3-1
Batch Manager, II:2-2
Batch utility, II:2-21
Batch, I:5-5, 5-21
Batch.run, I:5-1
Baud rate, I:8-4
Binding a system service, II:6-34
Blank space, on tape, II:8-7
Block size, II:8-17
Blocks, increasing size of, II:8-16
Buffer recovery order, II:8-12, 8-19
Buffer size, I:7-3
Buffers
  determining size of, II:8-8
  specifying size of, II:8-15
BuildAsyncRequest procedure, II:6-10, 6-23
BuildAsyncRequestDirect procedure, II:6-11, 6-23
Building request blocks, II:6-13
Byte streams
  spooler, II:3-3

c-characters, CD-ROM files, II:7-57
Call gate, I:2-12
Call progress tone detection (CPTD), II:4-24
Call progress tone detector, II:4-10
CdAbsoluteRead, II:7-3
CdAudioCtl, II:7-3, 7-21
  example, II:7-23
CDB, I:9-3
CdClose, II:7-3
CdControl, II:7-3
CdDirectoryList, , II:7-2, 7-8, 7-9
  example of, II:7-10
CdGetDirEntry, II:7-2, 7-17, 7-46
CdGetVolumeInfo, II:7-2, 7-4
CdOpen, II:7-3
CdRead, II:7-3
CdSearchClose, II:7-2, 7-12
CdSearchFirst, II:7-2, 7-11
CdSearchNext, II:7-2, 7-12
CdServiceControl, II:7-3
CdVerifyPath, II:7-2
CdVersionRequest, II:7-2, 7-12
CD-ROM
  character sets, II:7-56
  file formats, II:7-29
  files, example of, II:7-11, 7-12
  High Sierra directory record format, II:7-51
  High Sierra primary volume descriptor, II:7-39
  ISO directory record format, II:7-46
  ISO primary volume descriptor, II:7-30
  searching for files, II:7-11
  structures used, II:7-6
CD-ROM disc, specifying locations on, II:7-21
CD-ROM file, copying to disk, II:7-17
CD-ROM Service
  audio features of, II:7-21
  function of, II:7-1
  operations, II:7-2
  requirements for, II:7-1
ChangeCommLineBaudRate, I:8-4
Character
  attribute byte, I:3-22
  cell, I:3-2
  color, I:3-22
  coordinates, II:1-7
  cursor, II:1-7, 1-8
  map, I:3-3
  sets, for CD-ROM, II:7-56
CheckContextStack procedure, II:6-15
Child partition termination status, I:4-6

CleanQueue operation, II:2-8
Client operations
  for queue management, II:2-3
Client-server model, I:1-1
Clock source, I:8-4
CloseByteStream operation, II:3-3
Cluster network, I:1-1
Cluster server, II:8-1
ClusterCard, I:10-1
ClusterShare, I:10-1
CODEC (encoder/decoder), II:4-5, 4-8, 4-11, 4-12, 4-14, 4-15, 4-33
COED modules, II:6-33
Color intensity, I:3-6, 3-7
  three-palette format, I:3-8
Color programming, I:3-1
  character attribute byte, I:3-22
  color priority, I:3-25
  and graphics, I:3-4
  graphics colors, I:3-24
  palette control structure, I:3-9
  single-palette format, I:3-5
  three-palette color format, I:3-8
Command Descriptor Block (CDB), I:9-3
Command File Editor, I:5-6, 5-12
Command file, for installation, I:5-5, 5-12
CommLine interface, I:8-1
  Baud Rate, I:8-4
  clock source, I:8-4
  extensions, I:8-1
  reading signal status, I:8-4
  serial controller differences, I:8-3
  setting signal status, I:8-5
  using DMA with, I:8-5
Common-code module, II:6-29, 6-32, 6-36
  functions of, II:6-6
Communication controller, I:8-3
Communications DMA, I:8-5
  and DTR signal, I:8-5
  and WriteCommLineStatus, I:8-5
  External/Status interrupt, I:8-7
  getting status, I:8-8
  initializing, I:8-5
  Receive Special interrupt, I:8-7
  receiving data, I:8-7
  transmitting data, I:8-6
Communications Line Configuration Block, I:8-2
  fDMA field, I:8-6
  fV35Mode field, I:8-12
  fX21 field, I:8-11

Communications Line Return Area, I:8-2
  DMAAvailable field, I:8-6
  fV35Avail field, I:8-13
  ioX21 field, I:8-10
Compressing pauses, in voice files, II:4-15
Concepts, for programming a mouse, II:1-3
Config.sys parameters, I:1-16
Config.sys, I:3-13
ConfigureSpooler operation, II:3-1, 3-2, 3-7
Configuring the Queue Manager, II:2-9
Configuring the spooler, II:3-1 to 3-2
Connection handle, I:7-18
Conserving heap memory, II:6-16
Context Control Block (CCB), II:6-25, 6-26, 6-28
Context Manager, I:4-1, 4-5, 4-8, 5-13, 5-22; II:4-5
Context stack, II:6-15
Contexts, II:6-3
  managing, II:6-25
  other ways to use, II:6-28
  terminating, at deinstallation, II:6-28
Control file, for installation, I:5-5, 5-6
ConvertToSys, II:6-30, 6-33
Copying a CD-ROM file to disk, example of, II:7-17
CPTD configuration file, II:4-25
Create Message File command, I:5-11
CreateContext procedure, II:6-23, 6-26
Creating partitions, I:4-2
CSKNAMES.OBJ, I:10-1
  function definitions for, I:10-3
  kernel primitives supported by, I:10-2
  models of computation supported, I:10-3
CTOS, I:1-1
  accessing from DOS, I:10-1
  calling convention, where described, I:1-15
  development tools, I:1-2
  model of computation used, I:1-14
  protection model used, I:2-12
  SCSI Manager, I:9-1
  system calls, I:1-4
  system debugger, I:1-16
  system software, I:1-3
  use of call gates, I:2-12
  use of GDT-based segments, I:2-8
CTOS.lib version consistency, I:1-16
CTOS/XE, I:7-1
  exchanges and user numbers, I:7-2
  ICC buffer blocks, I:7-3
  standard connection handle, I:7-18

Cursor
  character, II:1-7, 1-8
  graphics, II:1-6
  movement of, II:1-15 to 1-16
  tracking of, II:1-7
Cyclic Redundancy Check (CRC), I:8-7

d-characters, CD-ROM files, II:7-56
d1-characters, CD-ROM files, II:7-57
Data, reading and writing, II:4-30
Data and voice
  separate lines for, II:4-21
Data blocks, on tape, II:8-7
Data call
  accepting, II:4-29
  converting a voice call to, II:4-29
  example, II:4-28
  originating, II:4-30
  starting, II:4-29
  terminating, II:4-30
Data Control Structure, II:4-29
Data management, II:4-2, 4-3
Data segment (DS) space, II:6-3
Data storage characteristics
  of sequential access devices, II:8-4
Data Terminal Ready (DTR), I:8-5
DDS devices, II:8-17
  recording data on, II:8-6
Deallocating heap memory, II:6-15
Debugger, I:1-16
Debugging
  aids, II:6-22
  an asynchronous system service, II:6-35
  statistics, II:6-23
Defining queues
  dynamically, II:2-17
  in the Queue Index file, II:2-14
  remote, II:2-23
DeinstallQueueManager operation, II:2-3, 2-8
Descriptor table, I:2-4
Development library consistency, I:1-16
Device routing, I:7-5
Devices, supported by Sequential Access Service, II:8-1
Dial characters, II:4-23
Dialer, II:4-34
Dialing telephone numbers, II:4-22
Digital data storage (DDS), II:8-1

Digital signal processor (DSP), II:4-4, 4-11, 4-12
Digitizing voice, II:4-5
Direct printing, II:3-1
Directory list buffer, for CD-ROM disc, II:7-8
Directory list, for CD-ROM disc, II:7-8
Disk
  activity, II:5-1, 5-5
  requirements, for voice files, II:4-16
  seeks, II:4-17
Distributed computing, I:1-1
DMA for communications. *See* Communications DMA.
DOS, I:10-1
  allocating exchanges under, I:10-2
  calling CTOS from, I:10-1
  identifying PC Emulator version from, I:10-4
  making CTOS requests from, I:10-2
DS (Data Segment)
  allocation, II:6-35
  space, II:6-15, 6-29, 6-33
DTMF
  encoder, II:4-5
  generator and receiver, II:4-5, 4-9
  tones, generating, II:4-22, 4-24

Early warning (EW), of sequential access devices, II:8-4, 8-8
Edf files, I:1-17
Editor, I:1-5
Electronic mail, II:4-2
End of data frame (EOF), I:8-7
End of Medium condition, II:8-9
Enhanced video, I:3-2
EnterDebuggerOnFault, I:1-16
Erase gaps, on tape, II:8-8
Escape sequences
  printer spooler, II:3-5
EstablishQueueServer operation, II:2-5, 2-8, 2-21
Exception, I:2-11
Executive, I:1-1, 3-4, 5-6
ExpandAreaSL operation, II:6-33
External/status interrupt, I:8-7, 8-10

Fault, I:2-11
fBackgroundColor, I:3-13, I:3-15, 3-25
fDataBufRecoverable, II:8-9
File formats, for CD-ROM, II:7-3, 7-29

File handle, I:7-18
File suffix conventions, I:1-12
File system activity, II:5-1
File transmission, II:2-3, 2-5
Filemarks, II:8-20
  on tape, II:8-7
Filter service, II:6-37
Fixed-length records, II:8-16
Flat file structure, for CD-ROM disc, II:7-6
Floppy installation, I:5-2
  naming files, I:5-14
Foreground color, I:3-2
fSuppressDefaultOnOpen, II:8-19

Gaps, on tape, II:8-7
Gate descriptor, I:2-12
General protection fault, I:2-11
GetCommLineDMAStatus, I:8-8
GetQMStatus operation, II:2-8, 2-17, 2-18, 3-5
GetWsUserName operation, I:7-4
Global Descriptor Table, I:2-8
Global variables, II:6-1
Graphics and color, I:3-4
Graphics cursor, II:1-6
  changing, II:1-14 to 1-15
  defining, II:1-12 to 1-14
GraphicsColorEnabled, I:3-26
GraphicsEnabled, I:3-26
Gray-scale monitors, I:3-21

Half-inch
  reel-to-reel devices, II:8-17, 8-20
    recording data on, II:8-6
  reel-to-reel tape, II:8-1
Handle, I:7-18
Header files, I:1-17
Heap, II:6-3, 6-15, 6-29
  allocating and deallocating, II:6-15
  conserving, II:6-16
HeapAlloc procedure, II:6-15, 6-16, 6-31
HeapFree procedure, II:6-15, 6-16
HeapInit procedure, II:6-30
Helical scan recording, II:8-6

Hierarchical file structure, for CD-ROM disc, II:7-6
High Sierra standard, for CD-ROM, II:7-1
  directory record format, II:7-51
  primary volume descriptor, for CD-ROM, II:7-39
Hold, placing a telephone line on, II:4-21

I/O, on a Series 5000 workstation, II:4-11
InitAlloc module, II:6-35
InitCommLine, I:8-2
  and DMA, I:8-5
  selecting extended features, I:8-2
  V.35 protocol support, I:8-12
  X.21 protocol support, I:8-11
Initializing the mouse, II:1-6
Initiator mode, I:9-1
Input event, II:1-7
Input/output switches, on a Series 5000 workstation, II:4-13, 4-14
Inquiry command, I:9-3
Install CDROM Service command, II:7-1
Install Queue Manager command, II:2-2, 2-21
Install Sequential Access Service command, II:8-2
Installation
  database, I:5-3
  media, I:5-2
    organizing, I:5-18
  script file, I:5-5, 5-11
  scripts, tips for creating, I:5-27
  variables, I:5-4, 5-21
  restarting, I:5-25
Installation Manager
  file lists created by, I:5-24
  verify feature, I:5-8
Installing
  Mouse Service, II:1-1
  Queue Manager, II:2-1, 2-2
Intel documentation titles, I:2-1
Internationalized call progress tone detection, II:4-24
  example of, II:4-27
Interprocess communication (IPC), II:2-1, 6-2
Interrupt service routine
  and DMA, I:8-5
  and External/Status interrupt, I:8-7
  and Receive Special interrupt, I:8-7

Inter-CPU communication (ICC), , I:7-2; II:2-1
  buffer size, I:7-3
ISO-9660 standard, for CD-ROM, II:7-1
  directory record format, for CD-ROM, II:7-46
  primary volume descriptor, for CD-ROM, II:7-30

JCL files, I:5-1, 5-11
  examples of, I:5-23, 5-43, 5-51, 5-59, 5-70

Kernel, I:1-3
Keys, II:2-19

Library version consistency, I:1-16
Link command, I:1-6
Link V6 command, I:1-7, 7-9
Linker, I:1-7
List file, I:1-6
Loading a cursor, problems with, II:1-17
LoadInteractiveTask operation, I:4-5
Local Descriptor Table, I:2-7
Local routing, I:7-5, 7-6
Log file, I:6-1
  and Volume Home Block, I:6-3, 6-6
  chronological order, I:6-6
  format of, I:6-1
  for installation, I:5-9
  reading, I:6-5
  record header and trailer, I:6-1
  wraparound, I:6-4, 6-5, 6-6
  writing records to, I:6-2
  written by file system, I:6-3
LogAsync module, II:6-22, 6-35
Logging messages
  for debugging purposes, II:6-32
Logging session, for Performance Statistics Service, II:5-1, 5-8
  closing, II:5-9
  opening, II:5-8
  reading a log, II:5-9
Logical address, I:2-2
Logical Unit (LU), I:9-2
LogMsgIn procedure, II:6-32
LogRequest procedure, II:6-32
LogRespond procedure, II:6-32

Main module
  for an asynchronous system service, II:6-8
Make, I:1-2
Managing contexts, II:6-25
Map file, I:1-8
Marking queue entries, II:2-6
MarkKeyedQueueEntry operation, II:2-6, 2-8, 2-20, 2-22
MarkNextQueuedEntry operation, II:2-6, 2-8, 2-22
Master FP name table, I:7-5
MCommands, I:7-3
Memory
  addressing, I:2-2
  freeing leftover, II:6-33
Merge Command Files command, I:5-12
Message file, for installation, I:5-5, 5-11
Minute-second-frame (MSF) format, on CD-ROM disc, II:7-21
Mixing programming languages, I:1-15
Models of computation, I:1-14
Modem, II:4-2, 4-7, 4-11, 4-34
  asynchronous use of, II:4-30
Monochrome graphics, I:3-22
Motion rectangle, II:1-7 to 1-12
Mouse
  buttons, II:1-3
  examples of how to program, II:1-3
  initialization procedures for, II:1-6
  procedures, by function, II:1-2
  tracking, II:1-15
Mouse Services, I:5-6, 5-12
  definition of, II:1-1
MS-DOS, I:10-1. *See also* DOS.
Multiple queue servers, II:2-6
Multiple voice messages, in one file, II:4-19

Naming
  conventions, I:1-8
  floppy installation files, I:5-14
  tape installation files, I:5-17
Nationalization, I:5-11, 5-26
Network, II:4-3
Normalized screen coordinates, II:1-4 to 1-5
NotifyCM request, I:4-6
NULL pointer, I:2-11

Object file, I:1-6
Object module library
  for mouse, II:1-1
  version consistency, I:1-16
Object module procedure, I:1-3
Object modules, binding, II:6-34
Offhook, II:4-21
OpenByteStream operation, II:3-3
Operating system calls, I:1-3
Operator, II:4-3

Package, I:5-3
Paging, I:2-6
Palette, I:3-5
  alphanumeric vs. graphics, priority, I:3-25
  control structure, I:3-9
  sample, I:3-15
Paragraph, I:2-3
Parallel recording, II:8-6
Parameter list
  variable length, in PLM, II:6-12
Parity, II:4-7
Partition
  consequences of unsuccessful task load, I:4-5
  creating, I:4-2
  deallocating, I:4-3, 4-10
  initializing, I:4-3
  loading a task into, I:4-5
  type, I:4-3
Partition management, I:4-1
  Action-Finish and swapping, I:4-7
  and child termination, I:4-6
  sample program, I:4-11
  termination procedure, I:4-8
  use of termination requests, I:4-7
Password
  to print, II:3-2
Pause compression, II:4-15
  advantages and disadvantages of, II:4-16
PBX systems, II:4-9, 4-21
PC Emulator version port, I:10-4
PC-DOS, I:10-1. *See also* DOS.
Performance Statistics Service
  example of, II:5-10
  function of, II:5-1
Performance Statistics Structure, II:5-4, 5-16
Piecemealable requests, I:7-3

Pixel, I:3-2
Pixel count, II:1-6
Playback, of compressed voice files, II:4-15
PLog, I:6-1, 6-2
  record-processing algorithm, I:6-9
Pointing device, II:1-1
Porting to protected mode, I:7-1
Power failure, II:2-1
Pre-GPS spooler byte streams, II:3-3
Primary volume descriptor, for CD-ROM disc, II:7-4
Print command, II:3-4
Print Manager, II:2-2, 2-21
Print wheel change, II:3-5
Printer channel, II:3-1
Printer spooler escape sequences, II:3-5
Printing, II:2-3, 2-5
  spooled, II:3-1
  direct, II:3-1
Private installation, I:5-2
Procedural interface
  asynchronous, II:6-10
  synchronous, II:6-10
Procedure naming, I:1-12
Processing
  asynchronous, II:6-2
  synchronous, II:6-2
Processor activity, II:5-1
Program example, of an asynchronous system service, II:6-36
ProgramColorMapper, I:3-1
  control structure, I:3-4
  functions performed by, I:3-4
  and monochrome graphics, I:3-22
  single-palette format, I:3-5
  three-palette format, I:3-8
Programming languages, I:1-2
Protected mode
  address calculation in, I:2-3
  descriptor tables, I:2-7
  exceptions and faults, I:2-11
  features of, I:2-1
  introduction to, I:2-1
  run file, I:1-7
PSCloseSession, II:5-2, 5-9
PSGetCounters, II:5-2, 5-6
PSOpenLogSession, II:5-2
PSOpenStatSession, II:5-2, 5-3
PSReadLog, II:5-2, 5-9
PSResetCounters, II:5-2
Public installation, I:5-2, 5-8, 5-23
Pulse Code Modulation (PCM), II:4-18

QIC tape device, II:8-16
recording data on, II:8-5
Quarter-inch cartridge (QIC) tape, II:8-1
Queries, mouse-related, II:1-7
QueryVideo, I:3-12, 3-21
Queue, II:2-1
adding an entry to, II:2-3
defining, II:2-14
defining dynamically, II:2-17
examples of typical, II:2-14
format of, II:2-11
referencing, II:2-17
removing an entry from, II:2-4
type 1, II:3-5
Queue entry
files, II:2-1
format of, II:2-13
handle, II:2-4, 2-18
header, II:2-13, 2-28
marking, II:2-6
processing order of, II:2-12
reading, II:2-4
referencing, II:2-17
rescheduling and removing, II:2-6
size, calculating, II:2-13
unmarking, II:2-6
Queue file header, II:2-11, 2-25
Queue handles, II:2-18
Queue index file, II:2-21, 2-22
example of, II:2-16
sample entries, II:2-23
Queue management
operations, by function, II:2-3
operations, sequence for using, II:2-21 to 2-22
of spooler queues, II:3-5
Queue Manager, II:2-21, 3-1
configuring, II:2-9, II:2-10
deinstalling, II:2-2
installing, II:2-1, 2-2, 2-23
run files, II:2-2
using across the network, II:2-22
Queue manipulation operations
summary of, II:2-7 to 2-9
Queue names, II:2-17
Queue server, II:2-3
establishing, II:2-5
multiple, II:2-6
operations, II:2-5

Queue Status Block, II:2-4, 2-8, 2-13, 2-19, 2-30
Queue type, II:2-16
Q-channel, II:7-22

Raster coordinates, II:1-6
ReadCommLineStatus, I:8-4
ReadKeyedQueueEntry operation, II:2-4, 2-8
ReadNextQueueEntry operation, II:2-4. 2-8, 2-18, 2-22, 3-6
Real mode address calculation, I:2-2
Receive command, I:9-5
Receive Special interrupt, I:8-7
ReceiveCommLineDMA, I:8-7
Record size, II:8-17
Record/playback, typical sequence for, II:4-19
Recording data
  on half-inch tape devices, II:8-6
  on QIC tape devices, II:8-5
Recording density, II:8-4, 8-18
Recording rates, for voice, II:4-15
Recording voice, II:4-14
Records, Sequential Access Service, II:8-16
Recovering buffer data, II:8-9, 8-13
Red Book standard, II:7-21
Referencing queues, II:2-17
Remote processor memory, I:7-2
Remote queue
  accessing, II:2-22
  defining, II:2-23
Remote routing, I:7-5, 7-7
RemoveKeyedQueueEntry operation, II:2-4, 2-9, 2-20, 2-22
RemoveMarkedQueueEntry operation, II:2-6, 2-9, 2-22
RemoveQueue operation, II:2-9
Removing queue entries, II:2-6
Request, I:1-3
Request-based system service, II:6-2
Request blocks, building, II:6-13
Request primitive, II:6-10
Request routing, I:7-4
  across the cluster, I:1-2
  inter-board routing directives, I:7-4
Request Sense command, I:9-6
RescheduleMarkedQueueEntry operation, II:2-6, 2-9
Rescheduling queue entries, II:2-6
ResetVideoGraphics, I:3-12, 3-15
Residual data, II:8-10, 8-11
Restarting an installation, I:5-25
RestartLabel, I:5-25

Restore command, II:7-9
ResumeContext procedure, II:6-24, 6-27
RewriteMarkedQueueEntry operation, II:2-9
Routing by device specification, I:7-10
Run command, I:1-8
Run file, I:1-7
  for an asynchronous system service, II:6-35
  for a system service, II:6-8
Run file mode, I:1-7, 2-8

Scheduling queue, II:3-1
Screen coordinates. *See also* Virtual screen coordinates.
  normalized, II:1-4 to 1-5
  virtual, II:1-4 to 1-5
SCSI, I:9-1
  devices, II:8-19
  interfaces, II:8-1
SCSI Manager, I:9-1
  application guidelines, I:9-16
  Command Descriptor Block (CDB), I:9-3
  initiator mode, I:9-1
  target mode, I:9-1
SCSI target mode
  Abort message, I:9-13
  application guidelines, I:9-16
  Bus Device Reset message, I:9-13
  commands accepted, I:9-2
  deferred errors, I:9-8
  Disconnect message, I:9-13
  guidelines for use, I:9-16
  Identify message, I:9-14
  illegal transfer length, I:9-18
  Initiator Detected Error message, I:9-14
  Inquiry command, I:9-3
  introduction, I:9-1
  messages accepted from initiator, I:9-12
  messages generated to initiator, I:9-12
  Receive command, I:9-5
  receiving data from initiator, I:9-9
  remote initiator requirements, I:9-1
  Request Sense command, I:9-6
  Send command, I:9-9
  Send Diagnostic command, I:9-10
  sending data to initiator, I:9-5
  sense data format, I:9-7
  session shutdown, I:9-19
  Synchronous Data Transfer request handling, I:9-15

SCSI target mode *(cont.)*
Test Unit Ready command, I:9-11
transfer length, I:9-6, 9-10, 9-18
ScsiTargetDataReceive, I:9-9
ScsiTargetDataTransmit, I:9-5
Searching, for CD-ROM files, II:7-11
Security mode, II:3-4
for printing, II:3-2
Segment address, I:2-4
Segment descriptor format, I:2-9
Segmented addressing, I:2-2
in protected mode, I:2-5
Selector, I:2-4
format of, I:2-7
Send command, I:9-9
Send Diagnostic command, I:9-10
Separators, CD-ROM files, II:7-58
SeqAccessCheckpoint, II:8-3, 8-14
SeqAccessClose operation, II:8-2, 8-12
SeqAccessControl, II:8-2, 8-7, 8-14
SeqAccessDiscardBufferData, II:8-3, 8-10, 8-12
SeqAccessModeQuery, II:8-3, 8-8
SeqAccessModeSet, II:8-3, 8-12
SeqAccessOpen, II:8-2
SeqAccessRead, II:8-2
SeqAccessRecoverBufferData, II:8-3
SeqAccessStatus, II:8-2
SeqAccessVersion, II:8-4
SeqAccessWrite, II:8-3, 8-9
Sequential access devices
data storage characteristics of, II:8-4
model of, II:8-4
Sequential Access Service
data buffering, II:8-8
determining size of service buffers, II:8-9
devices supported by, II:8-1
function of, II:8-1
installing multiple, II:8-1
placement of in cluster, II:8-2
programming considerations, II:8-17
records, II:8-16
recovering buffer data, II:8-9
Serial controller, I:8-3
CTS signal and X.21, I:8-9
and DMA, I:8-6
External/Status interrupt, I:8-7
initializing for X.21, I:8-12
Receive Special interrupt, I:8-7
Series 5000 workstations, II:4-1, 4-4, 4-11
Serpentine recording, II:8-5

Server installation, I:5-2
Shared resource processor (SRP), I:7-1;II:8-1
  Administrative Agent, I:7-3
  device specification, I:7-10
  exchanges on, I:7-2
  and GetWsUserName, I:7-4
  ICC buffers, I:7-3
  inter-CPU communication on, I:7-2
  multi-instance system services, I:7-10
  porting real-mode applications, I:7-1
  programming guidelines, I:7-1
  request routing, I:7-4
  sample request.txt file for, I:7-7, 7-11
  special handle types, I:7-19
  and system services, I:7-1
  use of handles, I:7-18
  user numbers on, I:7-2
Single-palette color format, I:3-5
  advantages of, I:3-7
Sketching program, using a mouse, II:1-18
Small Computer Systems Interface (SCSI). *See* SCSI.
Source code files, I:1-5
Spawn, I:4-1
Speech synthesis, II:4-3
Spooled printing, II:2-1, 3-1
Spooler, II:2-5, 2-9, 2-11
  configuration file, II:3-4
  configuration of, II:3-1 to 3-2
  definition of, II:3-1
Spooler queue, II:2-14
  control, II:3-7
  scheduling, II:3-6
  status, II:3-6
Spooler Status command, II:3-1, 3-2
SpoolerPassword operation, II:3-2, 3-8
SRP. *See* shared resource processor.
Stack pointer, II:6-3
Stack sharing, II:6-3
Standard connection handle, I:7-18
Static RAM (SRAM), II:4-11
Statistics ID block, II:5-2, 5-4
Statistics session,
  closing, II:5-8
  opening, II:5-3
  for Performance Statistics Service, II:5-1
Subpackages, I:5-3
SuperGen Series 5000 workstations. *See* Series 5000 workstations.
SwapContextUser procedure, II:6-21
SwapInContext, I:4-2
Swapping requests, II:6-18

Symbol file, I:1-7
Synchronous data communication, I:8-1
  Baud Rate, I:8-4
  clock source, I:8-4
  extensions to traditional interface, I:8-1
  reading signal status, I:8-4
  selecting extended features, I:8-2
  serial controller differences, I:8-3
  setting signal status, I:8-5
  using DMA, I:8-5
  V.35 protocol support, I:8-12
  X.21 protocol support, I:8-8
Synchronous processing, II:6-2
  diagram of, II:6-5
Synchronous request procedural interface, II:6-10
System build, II:3-3
System configuration tips, I:1-16
System log file, I:6-1
  chronological order, I:6-6
  format of, I:6-1
  reading, I:6-5
  record header and trailer, I:6-1
  and Volume Home Block, I:6-3, 6-6
  wraparound, I:6-4, 6-5, 6-6
  writing records to, I:6-2
  written by file system, I:6-3
System requests, II:6-17 to 6-21
System service, I:7-10
  asynchronous model of, II:6-1
  binding, II:6-34
  and the SRP, I:7-1
System-common procedure, I:1-3
  and the GDT, I:2-8

Tape
  general layout of, II:8-5
  logical elements within, II:8-7
Tape installation, I:5-2
  naming files, I:5-17
  organizing files, I:5-20
Target mode. *See* SCSI target mode.
Telephone
  lines, II:4-9
  management, II:4-3

Telephone Service, II:4-1
  data call example, II:4-71
  dialing example, II:4-36
  voice memory playback example, II:4-60
  voice response system example, II:4-40
Telephone Service Configuration Block, II:4-6
Telephone Status command, II:4-2, 4-3, 4-31
  function keys, II:4-35
  screen, II:4-32
Telephone Status monitor program, II:4-3
Telephone Status Structure, II:4-6
Telephone unit, II:4-9, 4-34
  parts of, II:4-9
  versus telephone line, II:4-21
TerminateAllOtherContexts procedure, II:6-29
TerminateContext procedure, II:6-28
TerminateContextUser procedure, II:6-19 to 6-20
TerminateQueueServer operation, II:2-6, 2-9
Termination requests, I:4-7, 4-18; II:6-17, 6-19
Test Unit Ready command, I:9-11
Three-palette color format, I:3-8
Timer Request Block (TRB), II:6-26, 6-28, 6-29
Timers, II:6-37
Track
  number, on CD-ROM disc, II:7-21
  definition of, II:8-5
Tracking the mouse, II:1-15
Transfer length, I:9-6, 9-10
TransmitCommLineDMA, I:8-6
Transport speed, II:8-18
Troubleshooting, programs that use the mouse, II:1-17
TsConnect operation, II:4-4, 4-5
TsDataChangeParams operation, II:4-7
TsDataCheckpoint operation, II:4-7
TsDataClose operation, II:4-29
TsDataCloseLine operation, II:4-7
TsDataOpenLine operation, II:4-7
TsDataRead operation, II:4-7
TsDataRetreiveParams operation, II:4-7
TsDataUnAcceptCall operation, II:4-7
TsDataWrite operation, II:4-7
TsDeinstall operation, II:4-5
TsDial operation, II:4-4, 4-22
TsDoFunction operation, II:4-5, 4-21
TsGetStatus operation, II:4-6
TsHold operation, II:4-6, 4-21
TsLoadCallProgressTones operation, II:4-6
TsOffHook operation, II:4-6, 4-21
TsOnHook operation, II:4-6, 4-21
TsQueryConfigParams operation, II:4-6

TsReadTouchTone operation, II:4-6
TsRing operation, II:4-6
TsSetConfigParams operation, II:4-6
TsVersion operation, II:4-6
TsVoiceConnect operation, II:4-4
TsVoicePlayBackFromFile operation, II:4-4, 4-20
TsVoiceRecordToFile operation, II:4-5
TsVoiceStop operation, II:4-5

Unmarking queue entries, II:2-6
UnmarkQueueEntry operation, II:2-6, 2-9

User configuration file, I:5-13
User number, I:7-2

V.35 protocol support hardware, I:8-12
Variable-length records, II:8-16, 8-18
VGA, I:3-2, 3-12
Video, during installation, I:5-13
Video Graphics Array. *See* VGA.
Video/Voice/Keyboard card (SGV-100), II:4-1
Virtual address, I:2-5
Virtual screen coordinates, II:1-4 to 1-5, 1-7, 1-8, 1-12
Voice amplifier, II:4-8
Voice and data, separate lines for, II:4-21
Voice Control Structure, II:4-4, 4-15, 4-19
Voice file, structure of, II:4-17
Voice File Header, II:4-18
Voice File Record, II:4-18
Voice management, II:4-2
Voice playback from memory, II:4-20
Voice Processor Module, II:4-1
  connections, II:4-10
  data features of, II:4-11
  voice features of, II:4-8
Voice
  recognition, II:4-3
  recording, II:4-14
  response system, II:4-10

Voice/Data Services
debugging using **Telephone Status** command, II:4-31
definition of, II:4-1
hardware features used by, II:4-8, 4-11
functional groups of operations, II:4-4
Volume control, on a Series 5000 workstation, II:4-13
Volume Home Block
and log file, I:6-3, 6-6

W-block, I:7-3
Wait loop, II:6-3, 6-4, 6-6, 6-25
Windows, II:1-7
Work area, for the Telephone Service, II:4-17
Workstations, character-mapped, II:1-6
WriteBsRecord operation, II:3-3
WriteByte operation, II:3-3
WriteCommLineStatus, I:8-5
Writing filemarks on tape, II:8-7

X.21 hardware
drivers-only mode, I:8-12
enabling and disabling, I:8-11
features of, I:8-9
X.21 protocol
general description, I:8-8
signal lines used, I:8-9
special signalling bit patterns, I:8-9
X.21 support, I:8-8
and External/Status interrupt, I:8-10
hardware features, I:8-9
initializing communications with, I:8-11
use of CTS signal, I:8-9
XE-530, I:7-1
XmitCommLineDMA, I:8-6

Y-block, I:7-3
Yacc, I:1-2

Z-block, I:7-3

USER'S COMMENT SHEET

CTOS Programming Guide, Volume I, General Programming Topics

09–02392

We welcome your comments and suggestions. They help us improve our manuals. Please give specific page and paragraph references whenever possible.

Does this manual provide the information you need? Is it at the right level? What other types of manuals are needed?

Is this manual written clearly? What is unclear?

Is the format of this manual convenient in arrangement, in size?

Is this manual accurate? What is inaccurate?

Name ______________________________ Date ____________

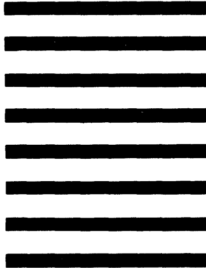
Title _________________________ Phone ________________

Company Name/Department _______________________________

Address ____

City ____________________ State _____ Zip Code ________

Thank you. All comments become the property of Unisys Corporation



BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 1807 SAN JOSE, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Unisys Corporation
Attn: Technical Publications
2700 North First Street
PO Box 6685
San Jose, CA 95150-6685

Fold Here

Tape Please Do Not Staple Tape