CTOS open

Programming Practices and Standards

Application Design

CTOS/Open
Programming Practices and Standards

Application Design

PRENTICE HALL
Englewood Cliffs, New Jersey 07632

Authors: *Alan Coleman* and *Margaret Morris*
Cover design: *April Bishop*
Page design: *Milena Martin-Arana*
Editorial/production supervision: *Mary Rottino*
Manufacturing buyers: *Kelly Behr and Susan Brunke*
Acquisitions editor: *Karen Gettman*

Published by Prentice-Hall, Inc.
A Division of Simon & Schuster
Englewood Cliffs, New Jersey 07632

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

ISBN 0-13-194382-0

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

Contents

Figures

Part I

Figures *(continued)*

Part II

Tables

Part I

Tables *(continued)*

Listings

Part I

Listings *(continued)*

Listings *(continued)*

Part II

Listings *(continued)*

About This Manual

This manual describes recommended techniques and practices for writing portable applications that follow the CTOS/Open standard for the CTOS® operating system. This manual is a companion volume to the *CTOS/Open Application Programming Interface Specification*.

The CTOS/Open standard is a specification which is implemented by several CTOS-based operating systems. The standard is not itself an operating system. When this manual talks about CTOS/Open, it means the specification. When it talks about CTOS, or about CTOS-based operating systems, it means those operating systems that conform to the CTOS/Open standard.

The CTOS/Open standard is a common subset of features. The operating systems that comply with it generally have additional features beyond the ones included in the CTOS/Open standard.

Purpose of This Manual

This manual describes practices and standards programmers should follow to ensure that their applications are portable across the various implementations of the CTOS/Open standard. Following the recommendations in the standard can greatly simplify porting an application from one CTOS-based operating system to another.

This manual can also serve as a "programmer's primer" for those new to CTOS/Open. It provides many examples, showing how to write commonly-used operations. This manual does not provide an in-depth conceptual explanation of CTOS implementation internals. For more comprehensive conceptual information, see the manuals for the CTOS-based operating system you use.

Audience

This manual is intended for programmers who want to write portable applications for CTOS-based operating systems. Programmers who are experienced with these operating systems, and programmers new to them, should both find the recommendations and examples in this book useful.

Related Documentation

The following documents may be useful to programmers who read this book.

CTOS/Open Books

- *CTOS/Open Application Programming Interface Specification*

  This document describes the procedural interface definitions for each of the procedures supported by the CTOS/Open Standard. It defines a common set of system interfaces that is consistent across all CTOS/Open-compliant versions of CTOS. The specification is intended for use by software developers and Independent Software Vendors (ISVs) who want to write applications that conform to the CTOS/Open Standard and are therefore portable across vendor platforms. It assumes that the reader has experience writing applications under CTOS or under another operating system.

- *CTOS/Open Application Programming Interface Specification: Computer Graphics Interface (CGI)*

  This document describes each CGI operation and explains how to write your own CGI programs.

- *CTOS/Open Application Programming Interface Specification: Graphical User Interface (GUI)*

  This document introduces an eXtensible Virtual Toolkit (XVT) and describes its relationship to other windowing interfaces. It also reviews the current XVT operation set, which includes functions,

macros, constants, and types. This specification is a virtual API and is intended for programmers who want to write applications that run in several different window environments and work across all CTOS-based platforms.

- *CTOS/Open Application Programming Interface Specification: Networking Services*

  This document describes the standard Link Layer interface used under the CTOS operating system. It defines the common requests used by all Link Layers and the requests used by the Data Link Manager. This specification also provides the standard format for reporting Link Layer status, and documents Link Layer error messages/codes, event codes, and commands. The final draft of this manual will include information on the Transport Layer.

- *CTOS/Open Application Programming Interface Specification: Printing Services (GPS)*

  This document describes how to write applications that use the Generic Print System (GPS) or the Generic Print Access Method (GPAM).

- *CTOS/Open Programming Practices and Standards: User Interface Design*

  This document provides guidelines for designing graphical user interfaces for CTOS-based applications. It contains information about the user interface standard call Common User Access (CUA), which has become part of the public domain. This manual is a working draft.

- *Exploring CTOS*

  This book gives a conceptual overview of the CTOS architecture. Additionally, it explains the advantages of developing applications on a distributed, message-based operating system.

Intel Manuals

- *iAPX 286 Programmer's Reference Manual*

  This book describes the features and instruction set of the Intel 80286 microprocessor.

- *80386 Programmer's Reference Manual*

  This book describes the features and instruction set of the Intel 80386 microprocessor.

How to Use This Manual

This manual is organized by function. Individual chapters are devoted to each of the most commonly-used functional areas of CTOS. To use this manual, examine the chapter or chapters that apply to your project, and make use of the examples and recommendations you find there. To look up specific topics, use the index at the back of the manual.

The CTOS Naming Convention

The examples in this book follow a specific naming convention, which is designed to promote the readability of source code. The underlying principal of the naming convention is this: use explanatory prefixes and suffixes on all variables and procedure names. Elementary programming practice dictates that the names themselves should also be explanatory. These conventions are particularly important when programming in a language like C, which tends toward cryptic syntax.

Each variable takes the form <Prefix><Root>Name<Suffix>. Prefix, root, and suffix do not all have to be present. In fact, most variables use only a prefix and a root in combination with the variable name itself. The following table describes the CTOS naming convention.

Table ATM-1. CTOS Variable-Naming Convention

(Page 1 of 2)

Token	Meaning
PREFIXES:	
b	Byte. A character or unsigned 8-bit integer.
c	Count. A two-byte unsigned integer.
f	Flag. A one-byte flag. True = 0FFh, False = 0.
i	Index. A two-byte unsigned integer.
l	Literal. A constant.
n	Number. A two-byte unsigned integer. Same as Count.
o	Offset. A two-byte offset from a segment base address.
p	Pointer. A logical memory address. Consists of a two byte segment identifier and a two-byte offset.
q	Quad. A four-byte unsigned integer.
rg	Array. Usually used with another prefix. For example, the prefix "rgb" identifies an array of bytes.
s	Size. A two-byte unsigned integer.
sb	String. An array of bytes where the first byte is the size of the string.
w	Word. A two-byte integer.
cb	Count of bytes.
pb	Pointer to a string of bytes.

Table ATM-1. CTOS Variable-Naming Convention

(Page 2 of 2)

Token	Meaning
ROOTS:	
erc	Two-byte status code.
exch	Two-byte exchange number.
fh	Two-byte file handle.
lfa	Four-byte logical file address.
ra	Two-byte relative address. Synonymous with offset.
rq	Request block. Size varies.
sa	Two-byte segment identifier.
sn	Selector. Segment identifier for a protected-mode memory address.
sr	Paragraph number. Segment identifier for a real-mode memory address.
userNum	Two-byte user number.
SUFFIXES:	
Last	Largest allowable index in an array.
Max	Maximum size of an array or buffer (Max = Last + 1).
Ret	Indicates a variable whose value is set by a called procedure and returned to the current one.

For example, to define a data buffer using this naming convention, we can assign four variable names:

pBuffer. A pointer to the start of the buffer.

sBufferMax. The maximum size of the buffer.

sBufferDataRet. The size of the data actually written to the buffer, returned by the procedure that writes to the buffer.

psBufferDataRet. Address of sBufferDataRet. Passed to the procedure that writes to the buffer, telling that procedure where to return the value of sBufferDataRet.

Part I – The Essentials

1

CTOS Overview

This chapter gives a brief overview of the major features of CTOS. It is intended for programmers who already have some familiarity with a CTOS-based operating system. For introductory information about your operating system, or for more in-depth conceptual information about it, see the manuals for your operating system.

What Is CTOS/Open?

The CTOS/Open Advisory Council (CTOS/Open for short) was formed as a joint effort among manufacturers, resellers, distributors, software developers, hardware developers, and users to establish and promote the CTOS-based architecture as a standard for distributed network computing.

The aim of CTOS/Open is to increase the number of CTOS-based applications available and to maximize the return on investment in software development for independent vendors and users. CTOS/Open sets portability standards for the CTOS operating system and its variants, and integrates evolving standards into a common, beneficial, and continuing strategy.

The Major Features of CTOS

CTOS has numerous features which, until recently, could only be found in operating systems designed for minicomputers and mainframes. This section describes some of the CTOS architecture's advanced features.

Multiprogramming

Multiprogramming is the ability of an operating system to run more than one program at a time. In a multiprogramming system, several programs are loaded into memory, and compete for processor time. The operating system manages the memory needs of each program, preventing the programs from overwriting each other. The operating system also ensures that each program receives a reasonable amount of processor time.

Multitasking

Multitasking, also called multiprocessing or multithreading, is related to multiprogramming. Multitasking is the ability of a single program to run more than one task at a time, similar to the way the operating system runs more than one program at a time.

In CTOS terminology, a program that performs more than one task is called a *multiprocess* program. Each of the threads of execution in a multiprocess program is called a *process*.

Process Scheduling

Any multitasking or multiprogramming operating system needs a method to determine which process should be running at any given point in time. CTOS uses an event-driven, preemptive priority scheduling scheme.

Each process is assigned a priority, with more critical processes (such as the operating system itself) receiving a higher priority than less critical processes. When a given process is running, it can be interrupted at any time by a higher-priority process. The lower-priority process cannot resume until all higher-priority processes have relinquished control of the processor (to wait for I/O, for example).

A process can never be interrupted by a lower-priority process. If no higher-priority processes are ready to run, the process will execute until it relinquishes control, no matter how many lower-priority processes are waiting.

As a general rule, the highest-priority process that is ready to run always has control of the processor.

The only exception occurs when a lower-priority process briefly disables the processor's interrupts to protect a critical section of code. A process should never disable interrupts for more than a few processor instructions.

Message-Based Operation

CTOS is a message-based operating system. The active programs and processes in a CTOS system communicate with each other and with the operating system by passing *messages*. Messages can be used to request services from the operating system and from other programs, to synchronize process execution, and to respond to requests for services. A message is simply a four-byte data item. Usually, the message is the address of a larger unit of data.

Messages are routed through *exchanges*. An exchange can be thought of as a mail box allocated by a process, but controlled by the operating system. When one process sends a message to another process, the operating system posts the message at the target process's exchange. The target process can then retrieve the message, and post a reply to the sending process's exchange.

Note that processes generally use messages to exchange the addresses of larger units of data. That data remains in the sender's memory, but is accessible to the receiver of the message. This allows very large data items to be exchanged with very little overhead.

The simple and flexible nature of CTOS messages allows them to be used for almost any type of interprocess communication. Also, unlike the subroutine calls used by many operating systems, messages can be intercepted before they arrive at their destination. This allows them to be modified or rerouted across a network without the sender's knowledge.

Extensible Via System Services

One of the primary advantages of the CTOS message-based architecture is that it allows the operating system to be "extended". As mentioned above, messages can be intercepted. Operating system messages are no different from application programs' messages, and can therefore be intercepted also.

This fact allows system programmers to write special programs, called system services, that either implement new types of messages, or filter and modify operating system messages. A system service can thus add new feature areas to the operating system, or change the operation of existing ones.

Further, system services can be dynamically installed and deinstalled. This allows the operating system to be modified or enhanced, then restored to its original state easily and without disruption.

Nationalization

Native language support (NLS) provides a set of utilities, run-time libraries, and data structures that can be used for the easy porting of software to run in various languages. Certain kinds of information, such as keycap legends and date and time formats, has been removed from the operating system itself and placed into a configuration file. Nationalization requires only modifying the configuration file and message files; relinking the software is not necessary.

Important Concepts

The following sections describe several essential concepts for a working knowledge of CTOS.

Processes, Messages, and Exchanges

In a previous section, we discussed the message-based nature of CTOS. In the CTOS operating system, *processes* pass *messages* to each other via *exchanges*.

The passing of messages is completely transparent to most application programs, but it is helpful to understand the mechanism. Figure 1-1 shows the message-passing process.

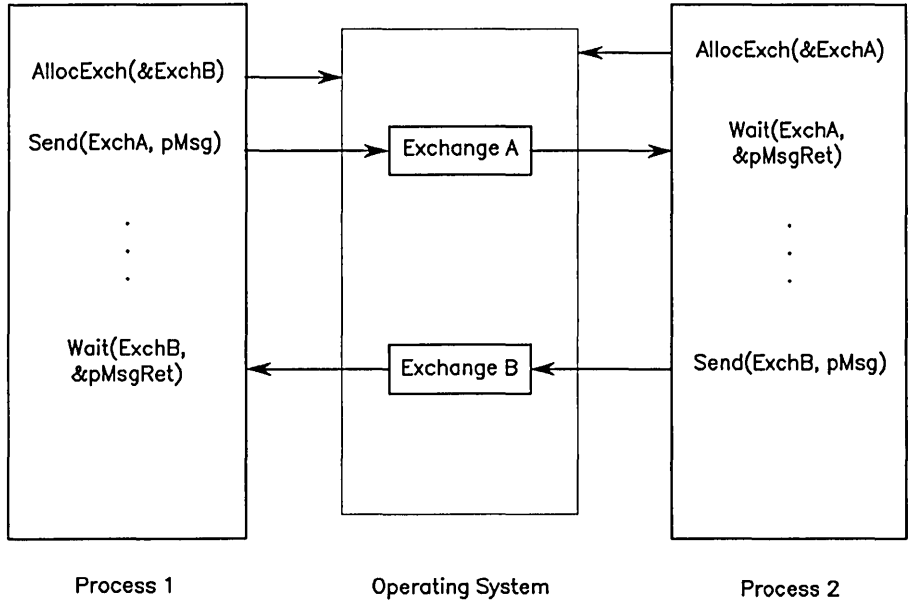


Figure 1–1. Message Passing

Each process in the figure allocates an exchange for its own use. It can then receive messages at that exchange. The two processes can send messages directly to each other's exchanges, if they know each other's exchange numbers. This knowledge is easy to acquire if the two processes are part of the same program. Simply store the exchange numbers in an area of memory to which both processes have access.

If the two processes reside in different programs, however, they cannot discover each other's exchange numbers directly. In this case, the processes can communicate using the request/response model.

The Request/Response Model

One of the most important concepts in CTOS is the request/response model. Most communication between application processes and the operating system uses this model.

Processes should not send messages directly, unless they are part of the same program. Instead, they should use requests. Every type of request has a unique numerical code. When a process wants to serve a particular request code, it notifies the operating system. The operating system then routes all requests which contain that request code to the process that volunteered to serve them.

The operating system maintains a table of all the request codes in the system, and of the exchanges at which they are served, so that it knows how to route each one correctly.

Figure 1-2 shows a diagram of the request/response model.

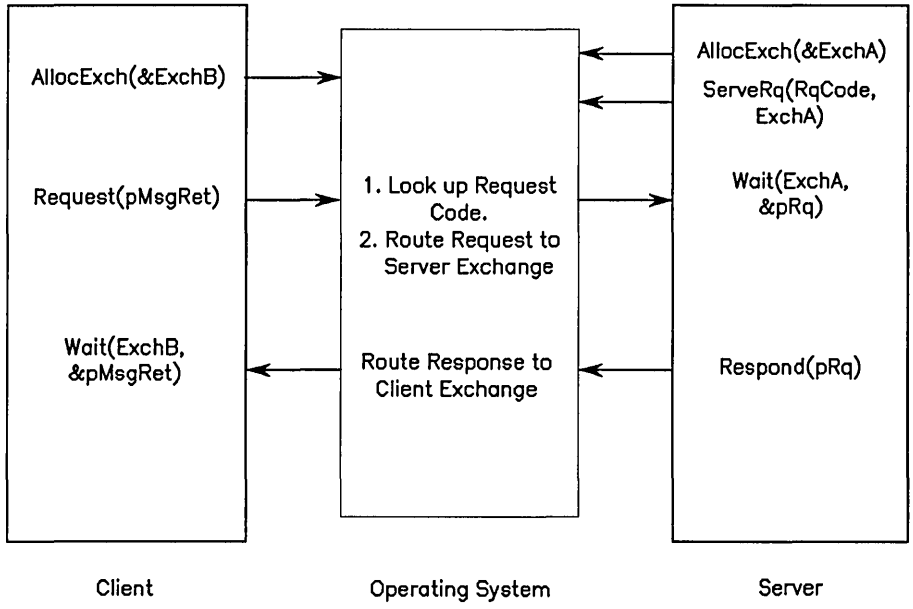


Figure 1–2. The Request/Response Model

The client application in the figure needs a service provided by another process. It therefore sends a request to the operating system for that particular service, and waits for a response. The operating system routes the request to the process which serves that request. The server of the request may be the operating system itself, or an appropriate system service.

The service provider performs the work required by the request, and sends back a response. The client application receives the response, and continues its execution.

Client programs can continue to execute while they wait for a response from a system service. Instead of simply waiting for the message to return, the client continues to do other work, but checks periodically to see if the message has returned.

Programs and Partitions

A *program* consists of code, data, and one or more processes. A *partition* is the area of memory in which a program runs. Partitions are longer-lived than programs.

When a partition is first created, it receives control of certain areas of memory. The operating system also loads a program into the new partition. Usually, this program is a user shell, such as the Executive, though the operating system can load an application program directly.

Any series of programs can then run in the partition, one after the other. Usually, the user returns to the shell between applications, although this is not required. Multiple programs can also run in a partition simultaneously. The partition itself continues to exist either until the system is rebooted, or until it is removed by a partition-management program, such as the Context Manager.

The Distributed Environment

The message-based nature of CTOS is uniquely suited to a distributed computing environment, and the hardware that runs CTOS-based operating systems has networking built in.

The building block of the CTOS distributed environment is the *cluster*. A cluster consists of one *master* and any number of cluster workstations. The master functions as a server for the cluster workstations, and is connected to them by a local area network. All workstations, including the master, can be used by people for their individual work.

When an application on a cluster workstation issues a request, its operating system serves the request locally if possible. If no process on the workstation serves that request code, the operating system automatically routes the request to the master for processing, unless the programmer defines the request as one that should not be routed.

The application does not know where the request is served. It simply issues a request, and receives a response. Network routing is completely transparent to it.

System Memory Organization

The operating system treats system memory as one large free area, into which it loads programs. There are certain rules for how it loads these programs, however.

Figure 1-3 shows the memory organization in a sample system. Portions of the operating system reside at both the highest and the lowest addresses in physical memory. Each program is then loaded at the highest available portion of memory. So, as system services and other programs are loaded, the range of available memory shrinks toward the lower addresses.

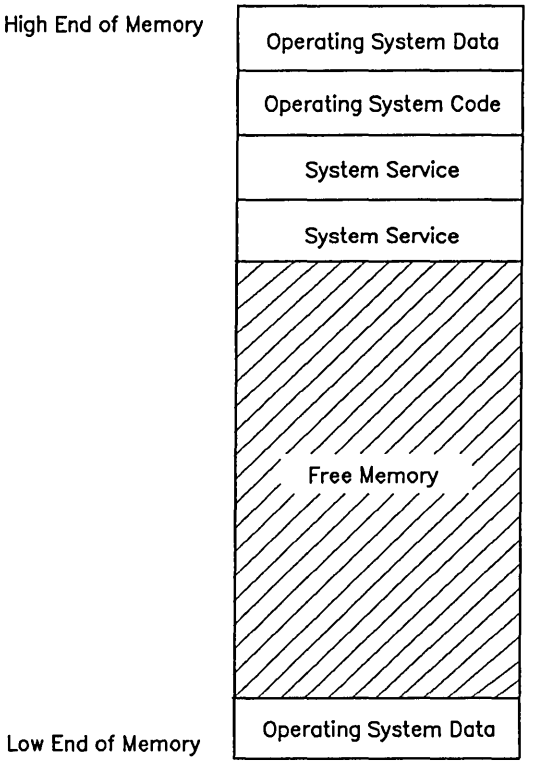


Figure 1–3. System Memory Organization

Application Memory Organization

As described above, programs run in partitions. A partition is an area of memory which the operating system has reserved for the use of a particular program, or succession of programs.

Each program is subdivided into *segments*. A segment is an area of memory which contains a certain type of information used by the program. Some segments contain code, others constants or static data, and one contains the program's stack. Figure 1-4 shows an example of how an application partition is organized.

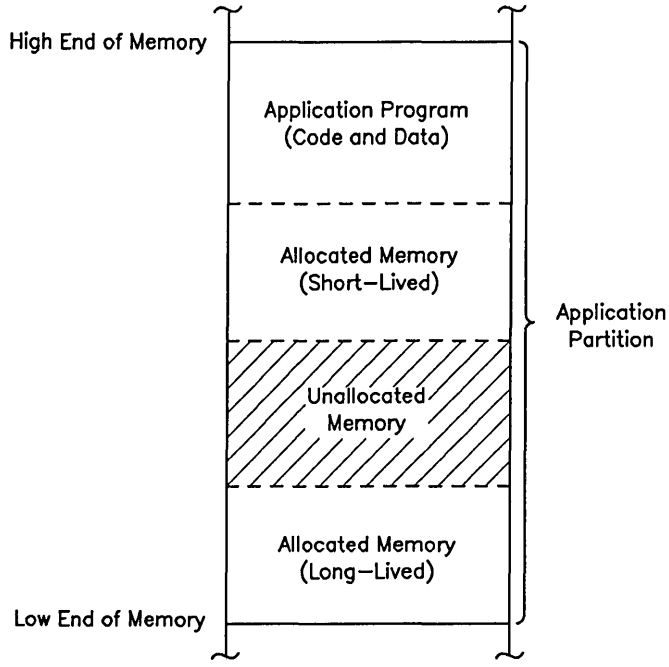


Figure 1–4 Application Partition Organization

A program rarely fills its partition completely at the moment it is loaded. Therefore, the partition contains some amount of unallocated memory, which the program can allocate and release dynamically. CTOS defines two kinds of allocatable memory; short-lived and long-lived.

Short-lived memory is the type of dynamic memory available under most operating systems. The program can allocate and release it at will. When the program terminates, the operating system releases any short-lived memory that the program did not explicitly release.

Long-lived memory can also be allocated and released at will, but long-lived memory is handled differently when the program terminates. If a program does not release its long-lived memory, that memory remains allocated after the program terminates. This allows the program to pass information to a successor program in the same partition.

The Anatomy of a Program

Programs that run in a CTOS operating system share certain characteristics. The following sections highlight several aspects that are common to all programs under CTOS.

Segmentation

All programs are divided into segments. Unless your program is written in Assembly language, your compiler generates code to set up your program's segments automatically.

At the minimum, a program has one segment which contains its code, and one which contains its constant data, static data, and stack. It may also use other, dynamically allocated data segments.

Most programs do not fit the minimum configuration, however. More commonly, a program has several code segments, a constant data segment, one or more static data segments, and a stack segment. In addition, most programs make use of dynamically allocated memory.

There is no required order for segments under CTOS, but DGroup (which normally contains constant data, static data, and the stack) must be separate from the program's code.

Figure 1-5, on page 1-12, shows the organization of a sample program in memory.

Types of External Procedures, and How to Call Them

CTOS provides several different types of external procedures, but an application can call them all in the same way. The types of procedures are:

- object module procedures
- system-common procedures
- request procedural interfaces
- kernel primitives

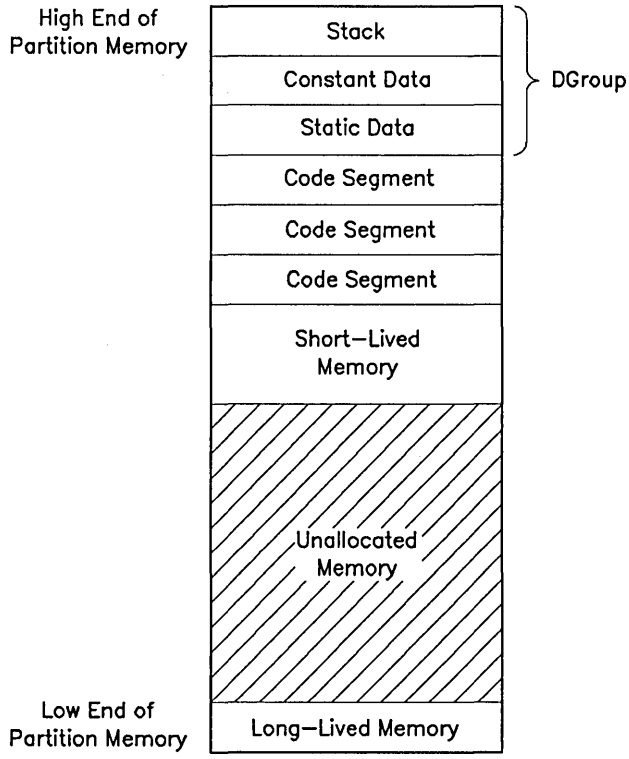


Figure 1–5. Segment Organization in an Application Partition

All these types of procedures can be called as external procedures from an application, but what actually happens in the operating system is different for each. For the purposes of this manual, they can all be treated as external procedures.

Listing 1-1 shows an example of an external procedure definition, and of a subsequent call to that procedure in the C language. If you use a different programming language, the syntax may differ, but the information is similar.

Note the two lines that begin with the *pragma* statement. These lines are required only because the C language uses a different parameter-passing convention than CTOS does. Some C compilers also provide a header file that contains these definitions. See your compiler manual for information about your compiler's calling convention. For more information about the calling convention used by CTOS, see "Stack Format and Calling Conventions" in Part II of this manual.

```
pragma Calling_convention(CTOS_CALLING_CONVENTIONS);
extern ErcType AllocAreaSL(Word cbArea , Pointer
                           ppAreaRet);
pragma Calling_convention();
.
.
.
erc = AllocAreaSL(cbMyArea, &pMyArea);
```

Listing 1–1. Declaring and Calling a CTOS External Procedure

A Note About pbcbs

Many of the CTOS/Open operations require the calling program to pass the address of some data item, and the size of that data item. These two parameters can be thought of as a pair, and are often called a *pbcb*, which is an abbreviation for "pointer to bytes and count of bytes". A pbcb defines an area of the calling program's memory which is made available to the called operation.

Two basic types of pbcb exist: request pbcbs and response pbcbs. For both types, the calling application passes a pointer to an area of its memory and the size of that area. The difference between the two types lies in who reads the information pointed to by the pbcb.

For example, a request pbcb might consist of a pointer to the beginning of a character string and the count of characters in the string. A response pbcb might consist of a pointer to an area of memory allocated for a returned string, and the size of that area (the maximum count of characters that can be returned).

So, in both cases the calling application passes a pointer and a count of bytes, which together define an area of memory. For a request pbcb, the

called operation reads the contents of that area of memory, and takes appropriate action. For a response pbcb, the called operation writes return information to that area of memory. The program then reads the returned information and takes appropriate action.

What Happens When a Program Starts

When a user starts a program, the operating system loader loads the program into short-lived memory, and starts it. The operating system allocates any memory that is needed at the start of the program, and makes the appropriate segment information available to the program.

Unless you write your program in assembly language, your compiler generates code that sets up your program's environment, then passes control to your main procedure.

The memory allocated by the operating system may be located anywhere in the machine's physical memory. There is no guarantee that any two segments are adjacent. This allows the operating system to make the most efficient use of its available memory. Generally, however, the operating system loads each program at the highest available address in physical memory.

The operating system also builds a structure called an Application System Control Block (ASCB), which contains the information the operating system needs to know about your program. The ASCB also contains information that the program needs, such as the location of its input parameters. The program can use various procedure calls to retrieve information from the ASCB. A program generally should not read directly from the ASCB, however, and should never write directly to the ASCB.

What Happens When a Program Exits

When a program terminates, control passes to the program's *exit run file*. By default, a program's exit run file is the environment (such as the Executive) from which it was launched. The program can change the exit run file to the name of another program, however, using the SetExitRunFile operation. This allows a series of programs to execute one after the other in the same partition.

The operating system also performs some housekeeping when an application terminates. When the operating system detects that a program wants to terminate, it notifies all the system services on the workstation. Those services can then deallocate any resources the terminating application was using, such as memory buffers or temporary files.

2
Basic Input and Output

This chapter describes how an application can read and write data, using the operations in the CTOS/Open standard.

Device-Independent I/O

CTOS provides both device-dependent and device-independent I/O. Device-independent I/O is easiest to use and most portable, but it has limited capabilities. Device-dependent I/O gives greater control over a given device, but limits the application to communication with a particular type of device.

For the purposes of this manual, device dependence refers to dependence on a type of device (such as video). It does not refer to dependence on a particular subtype of device (such as a bit-mapped monitor).

The Sequential Access Method (SAM)

The Sequential Access Method (SAM) provides a generalized, device-independent method of performing I/O. SAM allows a program to read input and write output in the same way, no matter what device the program is actually communicating with.

This lets you use the same subroutines to perform I/O to different physical devices, reducing the amount of new code a program requires. Because the SAM interface is consistent across device types, it also allows input and output to be redirected.

Compatibility comes at a price, however. SAM routines generally take more execution time than equivalent, device-dependent routines. Therefore, if execution speed is at a premium the device-dependent

routines may be more appropriate. In addition, if the application communicates with only a single type of device, the SAM routines require more memory than a single device-dependent procedure does.

What Is a Byte Stream?

The generalized method SAM uses for I/O is called a byte stream. A byte stream is exactly what its name implies: a series of bytes which a program reads or writes in sequential order. Byte streams can be used for input, for output, or for both. A byte stream must be opened before it is used, and should be closed when it is no longer needed.

An input byte stream can be read either until the reader chooses to stop reading or until the reader receives a nonzero status code (such as "end of file") in response to a read request. An output byte stream can be written until the writer chooses to stop, or until some physical boundary (such as a full disk) is encountered.

While all byte streams share a set of common procedures, SAM also provides a few device-dependent procedures for each type of byte stream. These procedures allow a small amount of device dependent control for different types of byte streams.

Each byte stream requires a 130-byte byte stream work area (BSWA). Some types of byte streams require an additional buffer area. A program can have any number of byte streams open at a time, but it must have a separate BSWA for each one.

Supported Byte Streams

CTOS supports most commonly-used byte streams automatically. You can also add new types of byte stream to your program, by customizing SAM. Customizing SAM is discussed in Part II, Chapter 17, "Building a Customized SAM." The byte stream device types supported by the operating system are listed below.

- [Vid] or [Vid]*n*

  The video byte stream. [Vid] specifies video frame 0. [Vid]*n* allows the program to specify any defined video frame. Video frames are

discussed below in "Video Management – VAM and VDM." Video byte streams are write-only.

The video byte stream is pre-opened by the operating system, and can be accessed using the external array variable bsVid as the BSWA.

- [Kbd]

  The keyboard. This byte stream includes input from submit files and batch files. The keyboard byte stream is read-only.

  The keyboard byte stream is pre-opened by the operating system, and can be accessed using the external array variable bsKbd as the BSWA.

- *{Node}[Vol]<Dir>filename*

  Any file identified by a full or partial file specification.

- [LPT]*&filespec*

  A Centronics-compatible printer connected to the parallel printer port. *&filespec* is optional, and specifies a configuration file for the printer.

- [PTR]*n&filespec*

  A serial printer attached to serial port *n*. As with the [LPT] device, *&filespec* is optional, and specifies a configuration file for the printer.

- [COMM]*n&filespec*

  Serial communications port *n*. *&filespec* is optional, and specifies a configuration file for the communications port.

- [QIC*m*]*n* or [TAPE*j*]*k*

  Quarter-inch cartridge (QIC) tape drive *m*, tape file *n*, or half-inch tape drive *j*, tape file *k*.

- *{Node}[QueueName]reportname*

  Spooler queue name. *QueueName* is the name of the scheduling queue for the spooler. The default is [SPL]. *Reportname* is a text string by which the entry is identified.

- [NUL]

  The system Null device. All input operations return status code 1 ("End of file"). All output operations return status code 0, but the output is discarded.

- [X25]*n&filespec*

  X.25 virtual circuit, where *n* is the network identifier. *&filespec* is optional, and specifies a configuration file that defines the virtual circuit's characteristics.

Device and File Specification Parsing

To determine the type of byte stream you want, SAM parses the device or file specification as follows:

1. It scans from left to right for an open-bracket symbol ('[').

   If SAM does *not* find an open-bracket, it assumes the string is a file name, and opens or creates a file with that name.

   If SAM *does* find an open-bracket symbol, it continues parsing the specification.

2. After it finds an open-bracket symbol, SAM tries to match the bracketed string to each of the reserved strings for system devices (for example, KBD).

   If SAM finds a match, it attempts to open the requested device.

   If SAM does not find a match, it continues parsing the specification.

3. Next, SAM checks for a left angle-bracket ('<').

   If SAM finds one, it assumes the string is a file specification, and opens a disk byte stream.

   If SAM does not find an angle-bracket, it assumes the string identifies a Generic Print System byte stream or a Spooler byte stream.

Using a Generic Byte Stream

Before using a byte stream, a program must open it. To do so, the program passes SAM the following information:

- a device or file specification for the byte stream
- a password, if required
- the byte stream mode (read, write, or both)
- the address of the byte stream's BSWA
- the address and size of the additional buffer, if required

After it opens the byte stream, the program can use it at will. The program should explicitly close the byte stream before exiting, however. If a byte stream remains open when a program exits, the operating system closes the byte stream, but does not check that all the program's data was written to it.

Listing 2-1 shows the body of a simple program that reads input from the keyboard, echoes it to the screen, then writes the input to a byte stream of the user's choice. It uses the procedure, GetByteStream, to open the byte stream. It then uses another procedure, GetString, to obtain a string of characters from an input device. It then writes the characters to the byte stream. Finally, if the byte stream is a file, the procedure ReplaceLineFeeds replaces all line feeds with page breaks. The GetString and ReplaceLineFeeds procedures are defined later in this chapter.

Listing 2-2 shows the variable declarations for the byte streams used in Listing 2-1. The entire program listing can be found in Appendix D.

```
GetByteStream (char *BSWA)
{
ErcType erc;
Word    cbRet;

do {
   CheckErc (WriteBsRecord (bsVid,rgbBsPrompt,
                            sizeof(rgbBsPrompt), &cbRet));
   cbString = GetString(bsKbd, rgbString, STRSIZE);
   if (cbString > 0) {
      erc = OpenByteStream (BSWA, rgbString, cbString,
                            NULL, 0, MODIFYMODE,
                            rgMyBuffer,sizeof(rgMyBuffer));
      if (erc == ERCNOTIMPLEMENTED)
         {  /* Mode modify not applicable, check write */
         erc = OpenByteStream (BSWA, rgbString,
                           cbString, NULL, 0, WRITEMODE,
                           rgMyBuffer, sizeof(rgMyBuffer));
         if (erc == ERCNOTIMPLEMENTED)
            {  /* Mode write not applicable */
            CheckErc (WriteBsRecord (bsVid, rgbInvalidBs,
                              strlen(rgbInvalidBs),&cbRet));
            CheckErc (WriteBsRecord (bsVid, rgbString,
                              cbString, &cbRet));
            CheckErc (WriteByte (bsVid,LINEFEED));
            erc = ERCBOGUS; /* set error to loop back */
            }
         }
      else CheckErc (erc);
      }
   else /* empty string, go get another one */
      {
      CheckErc (WriteByte (bsVid,LINEFEED));
      erc = ERCBOGUS; /* set error so we can loop back */
      }
   }
while (erc != 0); /* if they make a mistake, ask again */
}
main() {
ErcType erc;
Word    cbRet;

GetByteStream (MyBSWA);

/* read input with echo, then output it  */
/* single RETURN stops the input request */
do {
   CheckErc (WriteBsRecord (bsVid, rgbPrompt,
                            sizeof(rgbPrompt), &cbRet));
                                          continued...
```

```
    cbString = GetString (bsKbd, rgbString, STRSIZE);
    if (cbString > 0) {
       CheckErc (WriteByte (MyBSWA, LINEFEED));
       CheckErc (WriteBsRecord (MyBSWA, rgbString,
                                cbString, &cbRet));
       }
    } while (cbString != 0);

ReplaceLineFeeds (MyBSWA);
return(0);
}
```

Listing 2–1. A Sample Program Using Byte Streams

```
/* file access constants */
#define WRITEMODE     0x6D77 /* 'mw' */
#define READMODE      0x6D72 /* 'mr' */
#define MODIFYMODE    0x6D6D /* 'mm' */

/* allocation sizes */
#define STRSIZE       128
#define ALLOC_SIZE    0x100

/* input characters */
#define LINEFEED      0x0A
#define FORMFEED      0x0C

/* key definitions */
#define FINISH        0x04
#define CANCEL        0x07
#define RETURN        0x0A
#define Go            0x1B

/* error codes */
#define ERCOK         0x00
#define ERCEOF        0x01
#define ERCOPERATOR   0x04
#define ERCNOTIMPLEMENTED 0x07
#define ERCINVALIDBSWA    0x915
#define ERCBOGUS      0xFFFF

/* define the pre-opened byte streams */
extern char bsVid[];
extern char bsKbd[];
```

continued...

```
/* define my global variables */
char    rgbString[STRSIZE];  /* general purpose string */
char    MyBSWA[130];
int     cbString;
Word    rgMyBuffer[512];

char rgbBsPrompt[] =
     "\nType a device or file spec for the Byte Stream
     you want."
     "\nPress CANCEL if you make a mistake:  ";
char rgbExit[] =
     "\n\nPress GO to confirm FINISH, or CANCEL to
     continue.";
char rgbInvalidBs[] =
     "\n\nBytestream is invalid for output: ";
char rgbPrompt[] =
     "\n\nType a character string, then press RETURN:  ";
```

Listing 2–2. Variable Declarations for Byte Streams

Using Byte Streams for File Access

Byte streams are commonly used to read and write sequential data files. The program in Listing 2-1 could be used for that purpose.

Byte streams which access files also have two device dependent procedures available to them: GetBsLfa, and SetBsLfa. These procedures let a program determine and change the current Logical File Address (LFA) of a byte stream in a file. The Logical File Address is simply the offset, in bytes, of the current file position from the beginning of the file. The first byte in the file has an LFA of zero.

Listing 2-3 shows a procedure from the generic byte stream program. This procedure converts all line feeds in a file byte stream to form feeds.

```
ReplaceLineFeeds (char *BSWA) {
char    bIn;
ErcType erc;
long    Lfa;
/* flush my buffers */
CheckErc (CheckPointBs (BSWA));
erc = GetBsLfa (BSWA, &Lfa);
if (erc == ERCOK)    { /* it's a file */
   CheckErc (SetBsLfa (BSWA, 0));
   erc = ReadByte (BSWA, &bIn);
   while (erc == ERCOK) {
      if (bIn == LINEFEED) {
         /* change this to a form feed just for fun */
         CheckErc (GetBsLfa (BSWA, &Lfa));
         CheckErc (SetBsLfa (BSWA, (Lfa - 1)));
         CheckErc (WriteByte (BSWA, FORMFEED));
         }
      /* get the next character */
      erc = ReadByte (BSWA, &bIn);
      }
   }
else if (erc != ERCINVALIDBSWA)
   ErrorExit (erc);  /* something is wrong */

/* flush the buffer and close the file */
CheckErc (CloseByteStream(BSWA));
}
```

Listing 2–3. Positioning a Byte Stream in a File

Using Byte Streams for Video Access

Video byte streams have one device dependent procedure, QueryVidBs, which returns information about the current state of the video byte stream. However, video byte streams can also use escape codes to modify the state of the byte stream. Escape codes allow the program to perform the following functions:

- control screen display attributes
- control character display attributes
- control scrolling and cursor position

- control cursor visibility
- redirect the video byte stream to a different video frame
- enable or suppress a pause between screen-fulls of text

See Appendix D in the *CTOS/Open Application Programming Interface Specification* for a detailed description of the available byte stream escape sequences.

In addition, for simple byte stream video I/O, CTOS provides the OutputToVid0 procedure. This procedure outputs a text string to the current cursor position in video frame 0 (the default text frame), and is useful for displaying simple text messages.

Listing 2-4 shows a procedure that lets a user type an escape code, then sends that escape code to the video device. If the first character typed is an 'E', the rest of the string is interpreted as an escape code, otherwise the string is displayed as typed by the user.

```
PrintEscape (char *rgbEscape, char cbEscape) {
char     i, nCol, nFrame, nLine;
ErcType  erc;
Word     cbRet;

/* save the current cursor location */
CheckErc (QueryVidBs (bsVid, &strVidStatus));
nFrame = strVidStatus.nFrame;
nCol   = strVidStatus.nCol;
nLine  = strVidStatus.nLine;
/* change to frame 0 (working frame) */
CheckErc (WriteBsRecord (bsVid, escToFrame0,
                         sizeof(escToFrame0), &cbRet));
/* now print out the user entered escape sequence */
CheckErc (WriteByte (bsVid, ESCAPE));
for (i = 0; i < cbEscape; i++) {
   erc = WriteByte (bsVid, *(rgbEscape+i));
   if (erc == ERCINVALIDVIDESCAPE) {
      /* change to frame 1 (title frame) */
      CheckErc (WriteBsRecord (bsVid, escToFrame1,
                         sizeof(escToFrame1), &cbRet));
      /* print out error message */
      CheckErc (WriteBsRecord (bsVid, rgbInvEscape,
                         strlen(rgbInvEscape), &cbRet));
      return (-1);
      }
```

continued...

```
   }

/* output the sample string */
CheckErc (WriteBsRecord (bsVid, rgbTest, strlen(rgbTest),
                         &cbRet));

/* reset the frame just in case, can't use a canned */
/* sequence */
rgbEscape[0] = ESCAPE;
rgbEscape[1] = REDIRECT;
rgbEscape[2] = nFrame;
CheckErc (WriteBsRecord (bsVid, rgbEscape, 3, &cbRet));

/* reset the cursor postion */
rgbEscape[1] = COLSET;
rgbEscape[2] = nCol;
rgbEscape[3] = nLine;
CheckErc (WriteBsRecord (bsVid, rgbEscape, 4, &cbRet));
return (0);
}

main()
{
char    i, j;
char    *pString;
Word    cbRet;

ClearFrames ();
CheckErc (WriteBsRecord(bsVid, rgbIntro,
                        sizeof(rgbIntro), &cbRet));
/* get escape sequence, do it, ask for a new one */
do {
   i = 0;
   j = 0;
   CheckErc (WriteBsRecord (bsVid, rgbPrompt,
                            sizeof(rgbPrompt), &cbRet));
   cbString = GetString(bsKbd, rgbString, STRSIZE);
   if (cbString > 0)
       if (rgbString[i] == 'E')
         { /* we have the start of an escape sequence */
         i++;
         while (rgbString[i] != 0) {
            if (isalpha (rgbString[i]) != FALSE) {
               rgbEscape[j] = rgbString[i];
               i++;
               j++;
               }
            else if (isdigit (rgbString[i]) != FALSE) {
               pString = &(rgbString[i]);
                                                  continued...
```

```
                rgbEscape[j] = atoi (pString);
                j++;
                while (isdigit (rgbString[i]) != FALSE)
                   i++;
                }
             else i++;
             } /* end while rgbString[i] */

          rgbEscape[j] = 0; /* null-terminate it */
          PrintEscape(rgbEscape, j);
          } /* end if it starts with E */
       else CheckErc (Beep ());
    } /* end if strsize > 0 */
while (cbString != 0);

return(0);
}
```

Listing 2–4. Sending Escape Codes to Video

Using Byte Streams for Keyboard Access

There are no device-dependent procedures for keyboard byte streams, but keyboard byte streams return the following status codes when the **Finish** and **Cancel** keys are pressed.

- Finish (ASCII value 04) returns status code 1 ("End of file").
- Cancel (ASCII value 07) returns status code 4 ("Operator intervention").

Listing 2-5 shows a simple procedure that gets a string from the keyboard using a keyboard byte stream. Notice that the CheckErc procedure is not used when reading from a byte stream, because CheckErc causes the program to exit if it receives any status code other than zero.

```
GetString(char *BSWA, char *rgb, Word cbMax) {
char    bIn;
char    class;
char    i = 0;
ErcType erc;

/* zero the string */
while ((rgb[i] != 0) && (i < cbMax))
   rgb[i++] = 0;

/* collect the chars */
i = 0;
while (TRUE) {
   erc = ReadByte (BSWA, &bIn);
   if (erc == ERCEOF) HandleFinishKey (BSWA);
      /* Finish key pressed */
   else if (erc == ERCOPERATOR) return(-1);
          /* Cancel key pressed */
   else {
      switch (bIn) {
      case GO:
      case RETURN:
         return (i);
      default:
         /* check if it's a keyboard char */
         CheckErc (NlsClass (NULL, bIn, &class));
         if (class <= 2)
            {  /* valid character, echo it out */
            CheckErc (WriteByte (bsVid, bIn));
            rgb[i] = bIn;
            if (i < STRSIZE) i++;
            }
         else Beep ();  /* invalid character entered */
         break;
         } /* end of switch */
      } /* end of else */
   } /* end of while */
return(0);
}
```

Listing 2–5. A Procedure to Read a String from a Byte Stream

Other Uses for Byte Streams

Byte streams can be used for almost any type of I/O. Other common types of byte streams that this chapter does not discuss are printer I/O and communication port I/O.

In addition, you can create new instances of the standard byte stream procedure calls for special devices. For example, you could write byte stream procedures to communicate with a magnetic card reader, then link them with your program. The program could then use the standard byte stream interface for I/O, but use a device called [CardRdr], or something similar.

For more information on defining new byte streams, see Part II, Chapter 17, "Building a Customized SAM."

Device-Specific I/O

When the Sequential Access Method procedures are not able to perform the functions you need, you can use a second layer of I/O procedures. These procedures are more specific to the I/O device, and are therefore less portable.

File Management

CTOS provides a direct interface to the file system, as do almost all operating systems. Most programs use the byte stream interface unless they use a structured file format. In some cases, however, using the direct file system calls may be more efficient.

Listing 2-6 shows a procedure that opens a file using the direct file calls. The procedure first assumes the file exists and tries to open it. If the file does not exist, the procedure creates it. Listing 2-7 shows a procedure that writes to a previously opened file using the direct file calls. A listing for the program that contains both procedures can be found in Appendix D.

Note that the procedure in Listing 2-7 assumes the existing End-of-File pointer is divisible by 512. If it is not, the procedure will fail. Normally, as in the example, a program saves entire files at a time when it uses the direct file calls. Therefore, only the last call to the procedure results in

an End-of-File pointer that may not be divisible by 512: the true end of the file.

```
Word GetFile () {
ErcType erc;
Word    fh, cbRet;
DWord   qZero = 0;

do {
   CheckErc (WriteBsRecord (bsVid, rgbBsPrompt,
                            sizeof(rgbBsPrompt),&cbRet));
   cbString = GetString(bsKbd, rgbString, STRSIZE);
   if (cbString > 0) {
      erc = OpenFile(&fh, rgbString, strlen(rgbString),
                     NULL, 0, MODIFYMODE);
      if(erc == ERCNOSUCHFILE) { /* create the file */
         CheckErc(CreateFile(rgbString,
                             strlen(rgbString),NULL,0,0));
                                              continued...
         CheckErc(OpenFile(&fh, rgbString,
                           strlen(rgbString), NULL, 0,
                           MODIFYMODE) );
         }
      else {
         CheckErc(erc);
         /* reset file size and EOF pointer */
         CheckErc( SetFileStatus(fh, 6, &qZero,
                                 sizeof(qZero)) );
         CheckErc( ChangeFileLength(fh, qZero) );
         }
      }
   else { /* empty string, go get another one */
      CheckErc (WriteByte (bsVid,LINEFEED));
      erc = ERCBOGUS; /* set error so we can loop back */
      }
   }
while (erc != 0); /* if they make a mistake, ask again */
return(fh);
}
```

Listing 2–6. Opening a File

```
void StoreData(Word fhMyHandle, Word cBytes) {
DWord  qEOFPtr;
DWord  lfa;
Word   sDataRet;

/* get current lfa and EOF pointer */
CheckErc( GetFileStatus(fhMyHandle,0,&lfa,sizeof(lfa)) );
CheckErc( GetFileStatus(fhMyHandle, 6, &qEOFPtr,
                        sizeof(qEOFPtr)) );
/* increase file size */
lfa += 512;
CheckErc( ChangeFileLength(fhMyHandle, lfa ) );
CheckErc( Write(fhMyHandle, rgbMyBuffer, SMYBUFFER,
                qEOFPtr, &sDataRet) );
/* reset EOF pointer */
qEOFPtr += cBytes;
CheckErc( SetFileStatus(fhMyHandle, 6, &qEOFPtr,
                        sizeof(qEOFPtr)) );
}
```

Listing 2–7. Writing to a File

Wild Cards

CTOS provides two procedures to ease wildcard processing. CTOS defines two wildcard characters: '*' and '?'. The asterisk ('*') indicates that any string of 0 or more characters can be inserted at that position in the string and considered a match. The question mark ('?') indicates that any single character can be inserted at that position in the string and considered a match.

The Executive normally expands wildcards on a command form automatically, but if a program needs to read a file specification from another source, it should be prepared to handle wildcards. Listing 2-8 shows a simple routine that displays a list of files that match an entered file specification.

```
ErcType ListMatches (char *pFs) {
ErcType  erc = 0;
sdType   sdFs;
Word     *pBuffer;
Word     cbRet;

sdFs.pb = NULL;
sdFs.cb = 0;

if ((pBuffer = malloc (MEMSIZE)) == NULL)
   ErrorExit(ERCINSUFFMEM);

/* initialize the wild card expansion environment */
CheckErc (WildCardInit (pFs, strlen(pFs), pBuffer,
                        MEMSIZE));

while (erc == ERCOK) {
   /*get next file matching the wild card specification*/
   erc = WildCardNext (pBuffer, &sdFs);
   if (erc == ERCOK) {
      CheckErc (WriteBsRecord (&bsVid, sdFs.pb, sdFs.cb,
                               &cbRet));
      CheckErc (WriteByte (&bsVid, RETURN));
      }
   else if (erc == ERCEOF) return(0);
   else return (erc);
   }
}
```

Listing 2–8. Building a List of File Names from a Wildcard

Temporary Files

When a program makes use of temporary files, it should create them in the current volume's <$> directory. This directory is reserved by the operating system for temporary files. The operating system also performs a file name translation to prevent multiple instances of the same program from trying to use the same temporary file name.

When a program opens a file <$>Foo.tmp, the operating system expands the file specification to <$000>*nnnnn*>Foo.tmp, where *nnnnn* is the user number for that partition. This expansion is invisible to the program, so the program can always use the same temporary file name.

Because the user numbers at a cluster workstation are reassigned whenever the workstation is booted, you should never use the <$> directory for permanent files.

Keyboard Management

Using byte streams to read the keyboard is a relatively inefficient process. A program incurs all the byte stream overhead, but usually ends up reading one keystroke at a time. Using the keyboard-specific operations is more efficient and more flexible.

An application can read from the keyboard in either of two modes. The first, *encoded mode*, is the simpler one. In encoded mode, the application simply receives an ASCII character code whenever the user types one. The operating system makes the state of any special keys, such as Code or Shift, transparent to the application.

In *unencoded mode*, however, the application receives a *keyboard code* whenever a key is pressed or released. It is the application's responsibility to track and account for the states of any special keys. For example, the keyboard events that would return the character 'A' in encoded mode could return the following keyboard codes in unencoded mode:

- Left shift key depressed.
- 'A' key depressed.
- 'A' key released.
- Left shift key released.

Reading the keyboard in encoded mode is easier, but somewhat less flexible, than unencoded mode.

Listing 2-9 shows a modified version of the GetString procedure used in earlier listings. This procedure reads directly from the keyboard, in encoded mode. The procedure no longer uses byte streams, and now uses the TextEdit procedure to evaluate the entered character. TextEdit is a preferred method of processing keyboard input, since it handles most special keys for its client application.

```
/* read a string from the keyboard, echo it to a video
frame, and handle various special keys */
int GetString (char *str, int iFrame) {
char    bIn; /* the character */
char    i = 0;
ErcTypeerc;

/* set up buffer for TextEdit */
tbPtr = &textBlock;
textBlock.prgch= str;
textBlock.cchMax= STRSIZE;
textBlock.cchMac= 0;
textBlock.ichCursor= 0;

/* initialize string */
while ((str[i] != 0) && (i < STRSIZE))
    str[i++] = 0;

/* make cursor visible */
CheckErc (PosFrameCursor (iFrame, iCurCol, iCurLine));

/* get the chars */
while (TRUE) {
   CheckErc (ReadKbd (&bIn));
   switch (bIn) {
   case FINISH:
      PrintMsg (rgbExitString);
      CheckErc (ReadKbd (&bIn));
      if (bIn == GO) ErrorExit(0);
      else PrintMsg(NULL);
      break;
   case CANCEL:
      return(-1);
   case GO:
   case RETURN:
      /* make cursor invisible */
      CheckErc (PosFrameCursor (iFrame, ESCAPE, ESCAPE));
      return(textBlock.cchMac);
   case HELP:
      /* turn off cursor */
      CheckErc (PosFrameCursor(iFrame, ESCAPE, ESCAPE));
      DisplayHelp ();
      /* turn it back on */
      CheckErc(PosFrameCursor(iFrame, (iCurCol +
                        textBlock.ichCursor), iCurLine));
      break;
   case F1:
   .
   .
   .
```

continued...

```
    case F10:
       /* tell them they pressed a function key */
       PrintMsg (rgbFuncKeyMsg);
       CheckErc (ReadKbd (&bIn) );
       PrintMsg (NULL);
       break;
    default:
    /* add the char to the string if it's a valid char */
       erc = TextEdit (bIn, tbPtr);
       /* beep if it isn't */
       if (erc != 0) {
          CheckErc (Beep ());
          break;
          }
       /* display the modified TextEdit buffer, and move
          the cursor */
       CheckErc (PutFrameChars (iFrame, iCurCol, iCurLine,
                        textBlock.prgch, textBlock.cchMax));
       CheckErc (PosFrameCursor (iFrame, (iCurCol +
                          textBlock.ichCursor), iCurLine));
       break;
       } /* end of switch */
    } /* end of while */
return(0);
}
```

Listing 2–9. Procedure to Read a String from the Keyboard

Video Management – VAM and VDM

Video byte streams are the simplest video interface to use, but they allow little more than line-oriented output. The Video Access Method (VAM) and Video Display Management (VDM) allow a program to use multiple display frames, and to control how those frames appear on the screen.

Before a program can use VAM and VDM, it must set up its screen. To do so, the application must perform the following steps in the order shown below:

1. Call QueryVidHdw to determine the workstation's video capabilities.
2. (Optional) Call LoadFontRam to load a customized screen font.

3. Call ResetVideo to initialize the video control structures.

4. Call InitVidFrame to initialize each display frame the application will use.

5. (Optional) Call SetScreenVidAttr to set reverse video or half-bright video.

6. Call InitCharMap to initialize the character map.

7. Call SetScreenVidAttr to set the screen refresh attribute.

Listing 2-10 shows the standard setup procedures used for VAM and VDM. The program first resets the screen, then establishes its own screen display, consisting of several display frames. A program that uses this procedure can be found in Appendix D.

Note that the procedure uses the values returned by QueryVidHdw to set global values for the number of rows and columns on the screen. Note also that the procedure computes the number of lines in a frame based on those returned values. Because different hardware may support different numbers of rows or columns on the screen, these values should always be considered variable.

```
void InitVideo () {
Word    cb, sMap; /* counter, char map */
Byte    rgbVidHdw[3];

/* find out the screen size */
CheckErc (QueryVidHdw (&rgbVidHdw, sizeof(rgbVidHdw)));
nLines = rgbVidHdw[1];
nCols  = rgbVidHdw[2];

/* reset the screen */
CheckErc (ResetVideo (nCols, nLines, TRUE, 0x20, &sMap));

/* set the size of frame 0 */
sFrame0 = nLines - 6;

/* Initialize the frames */
CheckErc (InitVidFrame (MAINFRAME,0,3,nCols,sFrame0,0,
                        THINBORDERCHAR,0,0,0));
CheckErc (InitVidFrame (TITLEFRAME,0,0,nCols,3,0,
                        THICKBORDERCHAR,0,0,0));
                                          continued...
```

```
CheckErc (InitVidFrame (MSGFRAME,0,nLines-2,nCols,1,0,0,
                        0,0,0));
CheckErc (InitVidFrame (FKEYFRAME,0,nLines-3,nCols,3,0,0,
                        0,0,0) );
CheckErc (InitVidFrame (HELPFRAME,0,nLines-6 ,nCols, 3,0,
                        0,0,0,0));

CheckErc (InitCharMap (0, sMap));
CheckErc (SetScreenVidAttr (REFRESH, TRUE));

/* reset the frames */
CheckErc (ResetFrame (MAINFRAME));
CheckErc (ResetFrame (TITLEFRAME));
CheckErc (ResetFrame (MSGFRAME));
CheckErc (ResetFrame (FKEYFRAME));

/* display the title */
cb = sizeof (rgbTitle);
CheckErc (PutFrameChars (TITLEFRAME, ((nCols-cb)/2), 0,
                         rgbTitle, cb));

/* display the frame border */
cb = sizeof (rgbOut);
memset (&rgbOut,  THINBORDERCHAR, cb);
CheckErc (PutFrameChars (TITLEFRAME, 0, 2, rgbOut, cb));

/* display the function keys */
DisplaySoftKeys();
}  /*  end InitVideo */
/* Display the function key menu on the screen */
void DisplaySoftKeys ()
{
Word   cbOut, i;
struct FunctionKeyType
          {
          ByteFKeyStart;
          ByteFKeyCount;
          } rgCol[] = { {0,23}, {24,31}, {56,23} };

/* construct the function key display */
for (i = 0; i < 80; i++)
   switch (i)
      {
   case  7:
   case 15:
   case 31:
   case 39:
   case 47:
   case 63:
```

continued...

```
   case 7l:
      rgbOut[i] = VERTICALBAR;
      break;
   default:
      rgbOut[i] = rgbFkeys[i];
      }

/* display the function keys */
cbOut = sizeof (rgbOut);
CheckErc (PutFrameChars (FKEYFRAME, 0, 2, rgbOut,
                         cbOut));

/* display the frame border */
memset (&rgbOut, THICKBORDERCHAR, cbOut);
CheckErc (PutFrameChars (FKEYFRAME, 0, 0, rgbOut,
                         cbOut));

for (i = 0; i <= 2; i++)
   CheckErc (PutFrameAttrs (FKEYFRAME,
             rgCol[i].FKeyStart, 2,  HALFBRITEREVERSE,
             rgCol[i].FKeyCount));
}
```

Listing 2–10. VAM and VDM Setup Procedure

3
Error Handling Conventions

This chapter describes the error handling functions available under CTOS, and some common ways to use them.

Error Checking: General Practice

A general, recommended method of error checking under CTOS is the use of the CheckErc and FatalError procedures. CheckErc exits by calling FatalError whenever it encounters a nonzero status code. This makes CheckErc useful for general error checking when a program should abort in case of error.

If a program needs to take specific action when it encounters a particular status code, the program usually must implement its own procedure to handle that error code. Listing 3-1 shows an example of a procedure that checks for the end-of-file status code. It returns True if end-of-file is encountered, and False if no error is encountered. If it receives any status code other than zero or end-of-file, it calls FatalError to exit.

```
CheckEofErc(ErcType erc)
{
if(erc == ercOK)
    return(FALSE);
else if(erc == ercEOF)
    return(TRUE);
else
    FatalError(erc);
}
```

Listing 3–1. Procedure to Check for End-of-File

The FatalError Procedure and the fDevelopement Flag

FatalError allows the program to control its mode of termination. Most standard object modules distributed with CTOS call FatalError or CheckErc when they encounter errors. CheckErc also calls FatalError when it detects an error.

Normally, FatalError just calls ErrorExit with the encountered error code, but you can substitute your own procedure for it. For example, a system service might include a FatalError routine that causes the service to deinstall itself. This allows system services to call object module procedures without concern for the error handling within those procedures.

The FatalError procedure also checks a flag, fDevelopement (misspelled for historical reasons), before it terminates the program. If this flag is set to any value other than zero, FatalError enters the Debugger instead of calling ErrorExit.

To use the fDevelopement flag, just declare it as an external unsigned byte variable in your program and set it to TRUE. All calls to FatalError will then enter the Debugger instead of exiting.

Program Exit Modes: Exit, ErrorExit, ErrorExitString, and Crash

Each program must choose the appropriate exit mode when it encounters a severe error. In general, Exit should only be used after successful completion of a program's execution. When a program must exit on error, it should usually call FatalError, but can call ErrorExit or ErrorExitString directly if it needs to.

ErrorExitString works like ErrorExit, except that it allows the program to supply an information string along with the error code. The information string can be displayed or examined by the exit run file.

Crash should be used only when absolutely necessary. For example, after a system service is installed, it may need to call Crash if it detects an error which may affect other programs, and from which it cannot recover. This prevents the system from spreading the effect of the error to other

applications. Crash should only be used in those rare cases when not crashing the system could result in consequences worse than crashing.

For example, if a system service that filters calls to the file system encounters a severe error, the consequences could affect any program that opens a file. In this case, the service should probably call Crash.

Trapping Protection Faults

Under some circumstances, an application may need to trap any protection faults it causes. For example, a system service may want to enter diagnostic information in the system error log, then deinstall itself whenever it encounters a protection fault.

Also, an application program might want to trap protection faults and ensure that any critical information is written to disk before it exits. For example, the Editor traps faults in order to write its recovery typescript file to disk before exiting.

Listing 3-2 shows a procedure to set up a protection fault handler, and the procedure called by the fault handler. Note that any procedure called by a fault handler must exit in some manner. It can never return to the main program, or to the procedure that caused the fault.

Listing 3-3 shows the fault handler itself, which is written in Assembly language. Because they are trap handlers, all protection fault handlers must be written in Assembly language. For more information about trap handlers, see Chapter 19, "Interrupt Handlers" in Part II of this guide.

```
typedef void (*PF)();     /* pointer to function */
extern void _fault_handler(); /* our assembly function */
extern bsVid[];
PF p_fault_handler = 0; /* pointer to fault handler */
Pointer BadCSIP = NULL; /* used to store addr of fault */

/* set up a protection fault trap handler.  Input */
/* param is a pointer to the function that handles */
/* fault recovery or exit */
PF set_fault_handler(PF handler)
{
PF old_handler;
ErcType erc;

/* are we in protected mode? */
if ((FProtectedMode()) == FALSE )
    return 0;
/* save old fault handler so we can restore it later */
old_handler = p_fault_handler;
p_fault_handler = handler;
/* set the new handler as the erc 80 handler for this */
/* program */
if(erc=SetTrapHandler(13, (Pointer)_fault_handler) != 0)
    exit(erc);
return old_handler;
}

/* function to deal with protection fault */
void HandleFault() {
Word junklength;
Word erc;
char logmsg[] = "Erc 80 (Protection Fault) in Erc80.run";

/* tell 'em about it */
WriteBsRecord(bsVid, "Bad Pointer at CS:IP:  ", 23,
&junklength);
PutPointer(BadCSIP, 0xl0);
/* log the errror */
do {
    erc = LogMessage(&logmsg, strlen(logmsg) );
    if(erc == 290) /* buffer full erc */
        Delay(1); /* delay then retry */
    } while(erc == 290);
ErrorExit(80);
}
```

Listing 3–2. Setting a Protection Fault Handler

```
; FaultHandler.asm -- generic GP fault handler
extrn ErrorExit:far

DGroup group Data
assume ds: DGroup

Data segment word public 'Data'
;tell the Assembler about p_fault_handler
extrn p_fault_handler:WORD
extrn BadCSIP:WORD    ;tell it about CSIP pointer
Data ends

Stack segment stack 'stack'
    db  4   dup(?)
Stack ends

FaultHandler segment word 'code'
assume cs: FaultHandler
public _Fault_Handler
_Fault_Handler proc far
    mov ax, dgroup
    mov ds, ax
    ;load p_fault_handler
    mov ax, ds:word ptr p_fault_handler
    mov dx, ds:word ptr p_fault_handler + 2
    or  ax, dx              ;is p_fault_handler NULL?
    jz  exit                ;if it is, exit
    mov bx,sp               ;else
    mov cx,ss:[bx+2]        ;get CS:IP of fault
    mov BadCSIP,cx
    mov cx,ss:[bx+4]
    mov BadCSIP+2,cx
    ;then jump to fault
    jmp dword ptr p_fault_handler

exit:
    mov  ax, 80
    push ax
    call ErrorExit
_Fault_Handler endp
FaultHandler ends
end; FaultHandler.asm
```

Listing 3–3. Assembly Routine for Generic Trap Handler

Error Logging: WriteLog

Listing 3-4 shows a procedure that receives a text string as its parameter, then enters that string in the system error log. Because the error log buffer can become full, the calling procedure should check for that condition and take appropriate action. For example, the HandleFault routine in Listing 3-2 uses a delay loop to ensure that the message is always logged.

Error logging should be used for any situation in which an error may be difficult to trace. For example, if a system service encounters a severe error, and deinstalls itself as a result, it should log the occurrence. It should also log any additional information the user may need to determine the source of the error.

A program's entry in the system log must have a record type of 0FFF7h. This record type tells the system that the log entry is a text message. Using any other record type may result in an unreadable entry in the system log. WriteLog does *not* check for invalid user entries.

```
/* LogMessage - Routine that correctly logs a text string
*/

Word LogMessage(Pointer pbMessage, Word cbMessage)
{
ErcType erc;
struct {
    Word TextCode; /* value to indicate msg to Plog */
    Word cbText;
    Byte rgbText[120];
    } LogMsg;

LogMsg.TextCode = 0xFFF7; /* must always be 0xFFF7 */
LogMsg.cbText = cbMessage;
/* copy the message to the LogMsg structure */
memcpy(&(LogMsg.rgbText), pbMessage, cbMessage);
/* write the entry and return the erc */
return(WriteLog(&LogMsg, cbMessage + 4));
}
```

Listing 3–4. Procedure to Create a System Error Log Entry

4

Parameters and Command Form Processing

This chapter describes how an application can retrieve and interpret its initialization parameters when it is loaded into memory. This chapter also explains how to build a block of parameters for an application to pass to its successor.

The Executive

The Executive is the most widely available command-entry shell used with CTOS. The Executive lets users enter commands by typing the command name then filling out a menu which contains the command's parameters. A sample Executive menu might look like this:

```
Rename
   File To          _____________________________
   File From        _____________________________
   [Overwrite OK?]  _____________________________
   [Confirm Each?]  _____________________________
```

The user fills in the desired information on the menu, then presses **Go** to execute the command.

Command Forms and Parameters

Each input line on a command form constitutes one *parameter*. Some commands have no parameters, others have many. There is no specific limit on the number of parameters a command can have.

Each parameter consists of zero or more *subparameters*. If a line on the command form is left blank, that parameter has zero subparameters. Each entry on the line constitutes one subparameter. An entry consists of

a string of characters, either separated by spaces or enclosed in delimiters (single quotes).

The list below shows two examples.

File To Ralph Sampson Julie

This parameter has three subparameters: "Ralph", "Sampson" and "Julie".

File To 'Ralph Sampson' Julie

This parameter has two subparameters: "Ralph Sampson", and "Julie".

The Executive passes input parameters to the application program in a structure called a Variable-Length Parameter Block (VLPB).

While using an Executive command-form is the most common method of passing parameters to a program, there is no requirement to use it. Any program can pass start-up parameters to any other program by building a Variable-Length Parameter Block (VLPB). The rest of this chapter explains the general method of receiving and passing parameters with a VLPB.

The Variable-Length Parameter Block (VLPB)

The Variable-Length Parameter Block (VLPB) is a data structure in which parameters and subparameters are stored. The VLPB resides in the long-lived memory of an application partition. The program running in that partition can use procedural calls to retrieve parameters from the VLPB. It can also store parameters in the VLPB for a program to which it chains (its successor in the partition).

The address of the VLPB for a partition can be found in the partition's Application System Control Block, but it is rarely practical to write to the VLPB directly. The parameter management operations are much easier to use.

Conceptually, the VLPB is a two-dimensional sparse array of strings, indexed by row (iParam) and column (jParam). In terms of the Executive command form, the array is described by the following list.

- Each row in the array (iParam) corresponds to one line on the command form.
- Each column (jParam) in the array corresponds to one subparameter position. For example, when jParam is zero, it refers to the first subparameter in a given row.
- Each element in the array (iParam, jParam) consists of one subparameter typed on the Executive command form.

User parameters begin at parameter one. Parameter zero is reserved for system information. Parameter zero contains:

the command name at subparameter zero

the command case at subparameter one

the Redo keystroke buffer at subparameter two.

Figure 4-1 shows the contents of a VLPB for a sample Executive command.

rgParams (VLPB)	Subparameter (jParam) 0	Subparameter (jParam) 1	Subparameter (jParam) 2
Parameter (iParam) 0	Rename	00	Rename Fred Rogers Ralph Sampson
Parameter (iParam) 1	Fred	Rogers	
Parameter (iParam) 2	Ralph	Sampson	

Figure 4–1. Sample Variable-Length Parameter Block

Reading Input Parameters from the VLPB

A program can retrieve parameters from the VLPB using three operations: CParams, CSubParams, and RgParam.

First, the program should call CParams to determine the number of parameters in the VLPB. The program can then call CSubParams for each parameter, to determine how many subparameters each parameter has. Finally, the program can call RgParam to retrieve each subparameter.

Listing 4-1 shows a simple procedure for reading parameters from the VLPB. The procedure retrieves each subparameter from the VLPB, then prints the subparameter as a string of characters. The program that contains Listing 4-1 is in Appendix D.

Applications more often need to check individual subparameters for a particular value. They might use a procedure like the one in Listing 4-2. It checks a parameter to see if that parameter contains a "yes" or a "no", and sets a flag accordingly.

```
void ListParams () {
char        i = 0, fOut = 0;
ErcType     erc;
pbcbtype    pParam;
Word        cParam, cSubParam; /* count of params */
Word        iParam, jParam;/* parameter indices */

cParam = CParams (); /* get input param count */

/* scan through each parameter, forget parameter 0 */
for (iParam = 1; iParam < cParam; iParam++) {
  /* get the count of subparameters for this parameter */
   cSubParam = CSubParams (iParam);
   if (cSubParam > 0) {
      fOut = TRUE;
      printf("\n\nParameter %d has %d subparameter(s).",
             iParam, cSubParam);
      /* if the subparam count is > 0, print them */
      for (jParam = 0; jParam < cSubParam; jParam++) {
         erc =  RgParam (iParam, jParam, &pParam);
```

continued...

```
            /* quit if no parameter, break if no */
            /* subparameter */
            if (erc == ERCNOPARAM) return;
            else if (erc == ERCNOSUBPARAM) break;
            else CheckErc (erc);
            for (i = 0; i < pParam.cb ; i++)
               rgStr[i] = *(pParam.pb + i);
            rgStr[i] = 0;
            printf("\nThe entry in parameter %d,
                   subparameter %d is '%s'", iParam, jParam,
                   rgStr);
            }  /* end of for jParam */
         }
      }  /* end of for iParam*/

/* we have no parameters passed to us */
if (fOut != TRUE)
   printf ("\n\nNo parameters passed to %s.",
           pChainFile);
}
```

Listing 4–1. Reading a Variable-Length Parameter Block (VLPB)

```
/* is there input for this parameter? */
if( CSubparams(INCFLAGPARM) > 0) {
    /* get the first subparameter */
    CheckErc(RgParam(INCFLAGPARM, FirstSub, Paramptr));
    /* set flag for yes or no */
    CheckErc(NlsYesOrNo(NULL, Paramptr, &IncludeFlag);
    }
else
    /* default */
    IncludeFlag = TRUE;
```

Listing 4–2. Evaluating a Parameter in a VLPB

Creating a New VLPB for the Exit Run File

To pass parameters to a successor program in a partition, the current program must construct a VLPB for its successor. To do this, the program first resets the existing VLPB, then creates an area of memory for its new VLPB.

The program can then enter its parameters in its newly-created VLPB. When a parameter has only one subparameter, the program can use RgParamSetSimple to enter the value of that parameter.

When a parameter has multiple subparameters, though, the program must use RgParamSetListStart to begin the list of subparameters. It must then call RgParamSetEltNext for each subparameter entry.

Listing 4-3 shows a procedure for building a VLPB, and shows how the VLPB can be used to pass parameters to a successor program in a partition. This program passes only character strings, but numbers or pointers to areas of long-lived memory could also be passed in the VLPB.

The body of the program first resets the long-lived memory in the application partition. This deletes the previous VLPB.

Next, the program allocates memory for the new VLPB, and initializes it. Reinitializing the long-lived memory prevents the program from writing past the end of the existing VLPB and possibly destroying data.

The main program then calls GetNewParams, which fills in the new VLPB.

Finally, the program in the listing chains to itself. This causes it to execute again, using the new VLPB. The program that contains Listing 4-3 can be found in Appendix D.

```
void main() {
Pointer    pVLPB; /* used to build new VLPB */

/* initialize the video */
CheckErc (ResetFrame (EXECFRAME));
printf (pMsgIntro);

/* display all the parameters passed to me */
ListParams ();

/* reset long-lived memory, then allocate a new VLPB */
CheckErc (ResetMemoryLL ());
CheckErc (AllocMemoryLL (ALLOC_SIZE, &pVLPB));

/* Init the parameter block */
CheckErc(RgParamInit (pVLPB, ALLOC_SIZE, (MAXPARAMS-1)));

/* fill it in */
GetNewParams ();

/* tell the user I am chaining to myself */
printf (pMsgChaining);
CheckErc (Chain (pChainFile, strlen(pChainFile),
                 NULL, 0, PRIORITY, 0, FALSE));
}

void GetNewParams() {
char       bIn, fSkip;
int        cbStr;
pbcbtype   pParam;
Word       iParam = 1, jParam = 0;

while (iParam <= MAXPARAMS) {
   printf ("\n\nPress GO to enter parameter %d, RETURN to
            skip, CANCEL to Chain, or FINISH.\n", iParam);
   fSkip = FALSE;

   /* loop until we get one of the above characters */
   while (TRUE) {
      CheckErc (ReadKbd (&bIn));
      if (bIn == FINISH) ErrorExit(0);
      else if (bIn == CANCEL) return;
      else if (bIn == GO) break;
      else if (bIn == RETURN) {
         fSkip = TRUE;
         break;
         }
      else CheckErc (Beep ());
      }
```

continued...

```
   /* don't skip this parameter, collect subparameters */
   if (fSkip != TRUE) {

      /* initialize the subparam list for this param */
      CheckErc (RgParamSetListStart (iParam));
      while (jParam <= MAXPARAMS) {

         /* get the next subparameter */
         cbStr = GetString(rgStr, pMsgParam);
         if (cbStr > 0) {
            pParam.pb = rgStr;
            pParam.cb = cbStr;
            /* store it in the VLPB */
            CheckErc (RgParamSetEltNext (&pParam));
            jParam++;
            }

         /* we do not have a subparameter entry */
         else break;
         }  /* end of while jParam */
      }  /* end of if fSkip */
   iParam++;
   }  /* end of while iParam */
}
```

Listing 4–3. Building a Variable-Length Parameter Block (VLPB)

5

Protected Mode Programming Guidelines

This chapter describes the special programming requirements of the Intel protected mode architecture in the CTOS environment. This chapter provides guidelines that are sufficient for almost all applications. Where appropriate, this chapter refers you to additional sources of information.

Real Mode and Protected Mode Compatibility

Because protected mode is more restrictive than real mode, most programs that run in protected mode can also run in real mode. This chapter gives guidelines to help ensure that programs are compatible with protected mode addressing.

A full description of basic protected mode concepts is contained in the following Intel manuals:

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

Several of the compatibility issues presented are a result of differences in memory addressing between real and protected mode. This section assumes that you understand the addressing concepts of logical, linear, and physical memory addresses, which are described in the references cited above.

Emphasis in this section is on the logical memory address (commonly called a *pointer*). The pointer is what the programmer normally perceives when writing programs, while the physical address is used only by the processor. Pointer construction and manipulation are crucial to protected mode compatibility.

Review of Segmented Addressing

On Intel microprocessors, instructions do not accept physical addresses as operands; they accept only logical addresses (also called pointers). A *logical address* is formed from two parts, the *segment address* (SA) and the *relative address* (RA).

The SA identifies a region of memory (called a segment), while the RA identifies a particular byte within that region. The RA is often referred to as an *offset* from the SA. These addresses are usually written SA:RA (for example, 1E27:0000).

As each instruction executes, the processor hardware converts the two-part logical address to a physical address, which is then used to access physical memory. This translation is invisible to the program. No program can address physical memory directly. It can only use logical addresses.

Because every logical address is always relative to some segment address (SA), the Intel architecture is referred to as a *segmented addressing model.* Note the contrast to a *linear addressing model* such as the Motorola architecture, where instructions accept single, 32-bit linear addresses instead of SA:RA pairs. In the Motorola architecture, all addresses can be thought of as offsets from location zero in memory.

Real Mode versus Protected Mode Pointers

As discussed above, a pointer consists of a segment address (SA) and an offset (RA), and references a particular location in a program's memory space. The offset portion of a pointer is always the count of bytes from the beginning of the segment to the location referenced. The segment address, though, has different meanings in real mode and in protected mode.

In *real mode,* a segment address (SA) refers to a physical location in memory. Every sixteenth byte in memory has a segment address assigned to it. The segment address times 16 equals the physical address of that byte. For this reason, real mode segment addresses are sometimes referred to as paragraph numbers (16 bytes equals one paragraph in the Intel architecture).

In *protected mode*, a segment address refers only to an entry in a descriptor table maintained by the processor. The processor uses the segment address as a key to look up a physical memory address in the descriptor table. Because of this, there is no direct relation between the segment address and any location in physical memory. Protected mode segment addresses are sometimes referred to as selectors, to emphasize their function as a table lookup index.

In real mode, pointer arithmetic can be performed, because any pointer refers directly to some location in physical memory. In protected mode, however, no pointer ever refers to a location in physical memory. Instead, pointers refer to an index in a descriptor table, and to the offset from whatever location that index points to. Segment arithmetic or direct comparison of pointers is meaningless in protected mode, because any index in a descriptor table can point to any location in memory.

General Programming Guidelines

GUIDELINE: Do not use segment address arithmetic.

Compatible programs cannot add or subtract segment addresses to produce new segment addresses, because segment address arithmetic does not work in protected mode. (For reasons, see "Real Mode Versus Protected Mode Pointers," earlier in this section.)

GUIDELINE: Use FComparePointer to compare pointers.

Because multiple selectors may refer to the same physical memory location, protected mode programs cannot depend on the binary comparison of pointer values to test for pointer equality. (Pointers are equal if they reference the same byte location).

Compatible programs should use FComparePointer with a case value of 1 to test for pointer equality.

Example:

```
/* Compare the pointer equality of a response pointer */
/* to a request block pointer.*/
Pointer    pRqRet;
Word       Rq;
.
.
.
CheckErc(Wait(exch, &pRqRet));
if( FComparePointer(pRqRet, &Rq, 1) == TRUE) {
    /* The pointers are equal (that is, they reference */
    /* the same byte location).*/;
    }
```

Listing 5–1. Comparing Pointer Equality

GUIDELINE: For contiguous data objects larger than 64K bytes, use multiple segments.

A real mode program can make a single call to AllocAllMemorySL to allocate a large data object (more than 64K bytes), then address the data using segment address arithmetic. This technique fails in protected mode. (See "Real Mode Versus Protected Mode Pointers," earlier in this section for details.)

The recommended method for compatible programs is to make multiple calls to AllocMemorySL. This will decompose the large data object into multiple segments, each of which is then addressable by a returned pointer. The memory allocated by multiple calls to AllocMemorySL is contiguous.

GUIDELINE: Use compatible memory management operations.

Several memory management operations have been specifically designed for use by compatible programs. These are

AllocAreaSL
ExpandAreaLL
ExpandAreaSL
ShrinkAreaLL
ShrinkAreaSL

These memory operations manage memory allocation and deallocation within a single segment. For more information about these operations, see Chapter 16, "Memory Management" in Part II of this manual, or see the *CTOS/Open Application Programming Interface Specification.*

Compatible programs that depend on detecting contiguity between separately allocated chunks of memory should use these operations. They provide up to 64K bytes of contiguous memory. A program can test for contiguity within a segment by comparing pointer offsets (RAs).

Compatible programs should use ExpandAreaSL instead of AllocMemorySL if they make multiple calls to allocate small, fixed-length segments. This is because, in protected mode, each call to AllocMemorySL allocates a new selector, while ExpandAreaSL does not.

GUIDELINE: Use ExpandAreaSL to allocate DS-relative memory.

The Linker's data segment (DS) allocation option enables run time expansion of a program's static data area. The Linker assigns static data addresses at the high-order end of the data segment's 64K-byte range, leaving address space below for expansion.

DS allocation requires that static data be located at the low-order end of the program image. For more information about DS allocation, see Chapter 10, "Stack Format and Calling Conventions," and Chapter 16, "Memory Management" in Part II of this manual.

Compatible programs should use ExpandAreaSL instead of AllocMemorySL to dynamically expand the data segment. Programs that use AllocMemorySL will not work in protected mode because, instead of expanding the existing data segment, each call to AllocMemorySL allocates a new selector for a new segment.

GUIDELINE: Do not use the memory array option.

The Linker's memory array option directs the operating system to allocate additional memory at the end of a program's static data. If a program needs additional memory relative to its static data segment, it should use the DS Allocation option described above.

GUIDELINE: Do not use segment registers to store data.

In real mode, segment registers can be used to store data with no adverse consequences.

In protected mode, however, loading a segment register with a value that is not a valid selector will cause a protection fault, which is a fatal event in a program.

Compatible programs should use segment registers only for valid segment addresses.

GUIDELINE: Use proper NIL pointers.

A NIL pointer, also called a NULL pointer, is a pointer that has a segment address of zero.

Historically, when a pb/cb (pointer/length) pair was used, a cb of zero was often used to indicate a NIL pointer. This convention does not work in protected mode, because the arbitrary data in the uninitialized pb (pointer) may be loaded into a segment register *before the length is checked*. If this happens, it causes a protection fault.

Compatible programs should always initialize NIL pointers to zero.

Example:

```
Pointer     pRet;
Word        cbRet;
Pointer     p;
.
.
.
/* Call the InitPbCb procedure (defined later) to set */
/* the pointer to NULL.*/
InitPbCb(&pRet, &cbRet);

/* Use of a NIL pointer prevents a general protection */
/* (GP) fault if pRet is copied.*/
p = pRet;
}
```

continued...

```
/* Use a procedure (InitPbCb, in this example) to set */
/* the pb/cb to zero.*/
void InitPbCb(Pointer ppRet, Pointer pCbRet) {
*ppRet = 0;
*pCbRet = 0;
}
```

Listing 5–2. Initializing Pointers

GUIDELINE: Code segments should have CODE or COED for the name of the segment class.

In protected mode, the contents of a segment are executable as code only if the segment is typed as a code segment.

Compatible programs should use the class name CODE or COED for code segments. Most compilers use the CODE class name for code segments, though there are special cases. The COED class name is created by the Coed source build utility and is used to designate initialization code that is deallocated after use.

Make sure that code segments defined in assembly language programs have the class name CODE or COED.

Example:

```
;declare a code segment with name "myCode" and class
;"CODE"
myCode segment 'CODE'
    .
    .
    .
myCode ends
```

GUIDELINE: COED Segments

COED segments can be deallocated after the initialization code in them has been used. COED segments are simply code segments having the class name COED. This class name is created by the Coed source build utility.

COED segments must be located at the beginning of the run file image, immediately before static data. A program normally deallocates its COED segments using the DeallocMemorySL operation.

In real mode, the size of the memory deallocation is computed using segment address arithmetic. This does not work in protected mode.

Compatible programs using COED segments should use different deallocation schemes in real and protected mode. Deallocation should be based on a run time check of the execution mode using the FProtectedMode operation.

In protected mode, deallocate the COED segments by calling DeallocMemorySL for each COED segment using the following parameters:

pSegment — Is SN:RA, where SN is the COED segment selector, and RA is 0.

cBytes — Is 0FFFFh, meaning deallocate the entire segment.

The usual stack discipline rules apply to DeallocMemorySL. COED segment deallocation must occur before the program's first call to AllocMemorySL or AllocAreaSL. COED segments must be deallocated in the order of first defined (in other words, located lowest in the program's memory image), first deallocated.

For more information about disposable code, see Chapter 10, "Stack Format and Calling Conventions" in Part II of this manual.

GUIDELINE: Writable code segments are not permitted.

In protected mode, nothing can be written to a segment that is defined as a code segment. Therefore, compatible programs cannot have writable variables in the code segment, nor can they contain self-modifying code.

If you need to write to an executable segment, or need to make a writable segment executable, you must first create an alias for the existing segment. Then, you can use the SetSegmentAccess operation to change the segment type so that it may be written (or executed).

The following example creates an alias for the program's data segment. It then sets changes the segment type to CODE. If the array in the example

contained machine-language instructions instead of characters, they could be executed. Instead, the example just prints the array to verify that the alias is valid, then compares the original and alias pointers.

Example:

```
Pointer     pData;
Selector    sgCode;
Word        offset;

union{
    POINTERWords PtrStr;
    Pointer    ptr;
    } u;

char pTestMsg [] = "\"This is a test\"\n";
char pCodeMsg [] =
    "The string referenced by the code pointer = ";
char pDataMsg [] =
    "The string referenced by the data pointer = ";
char pDiffMsg []  =
    "\nAlias Code Pointer and Data Pointer reference
     different locations";
char pSameMsg []  =
    "\nAlias Code Pointer and Data Pointer reference the
     same location";

void main () {

/* get the offset of our Test Message array within DS */
pData = &pTestMsg;
offset = offsetof (pData);
printf (pDataMsg);
printf (pData);

/* now we want an alias based on the start of our DS */
offsetof (pData) = 0;

/* create the alias and define it as executable code */
CheckErc (CreateAlias (pData, &sgCode));
CheckErc (SetSegmentAccess (sgCode, 10));

/* now construct a pointer to our code and print the */
/* string */
u.PtrStr.ra = offset;
u.PtrStr.sa = sgCode;
printf (pCodeMsg);
printf (u.ptr);
```

continued...

```
/* finally compare data pointer and code pointer */
pData = &pTestMsg;
if (FComparePointer (pData, u.ptr, 1) != FALSE)
   printf (pSameMsg);
else
   printf (pDiffMsg);
ErrorExit (0);
}
```

Listing 5–3. Creating a Pointer Alias

GUIDELINE: Avoid timing loops.

Compatible programs should not contain timing loops or busy wait loops that depend on instruction execution speed. Use the Delay operation when the delay time is at least 100 milliseconds. Use the ShortDelay operation when the amount of time is less then 100 milliseconds.

Example:

```
/* delay 0.5 sec */
CheckErc(Delay(5));
```

GUIDELINE: Use only compatible instructions.

The 80186 and 80286 microprocessors provide extensions to the 8086 microprocessor instruction set. Similarly, the 80386 provides additional extensions.

A compatible program should use the instruction set of the least capable machine on which it is to run.

If a program needs to run on 80186-based computers, as well as 80286 and 80386 computers, do not use the PUSH SP instruction. When it executes that instruction, the 80186 pushes a different value than the 80286 and 80386 do.

Language-Specific Guidelines

GUIDELINE: **NIL pointer problems may occur as a result of certain coding sequences.**

In protected mode, it is valid to put 0 in a segment register, but not valid to use it to address data. A segment register with 0 in it is "out of action" temporarily. Of course, attempting to reference data at paragraph number 0 in real mode is usually an error as well, at least in a user program.

Constructs such as the following PL/M example, however, may be valid in real mode but cause a protection fault in protected mode, due to the kind of code generated by the compiler.

Example:

```
declare pFoo              pointer;
declare foo based pFoo    word;

if (pFoo <> 0) AND (foo = 1) then
    do;
        .
        .
        .
    end;
```

In PL/M, the preceding construct gives the compiler permission to test pFoo and foo in either order. Access to foo, however, must NEVER take place if pFoo is NIL, or a general protection fault will occur. To avoid this fault, the preceding code should be rewritten as follows:

```
declare pFoo              pointer;
declare foo based pFoo    word;

if pFoo <> 0 then
    IF foo = 1 then
        do;
        .
        .
        .
    end;
```

GUIDELINE: Check the Execution mode.

A compatible program may have good reason to use different algorithms in real and protected modes. Such programs may still be packaged as a single run file by including both algorithms with run time checks in the code to choose between them.

Compatible programs should use the FProtectedMode operation to determine their execution mode.

Example:

```
if( FProtectedMode() != FALSE)  {
    /*Use a protected mode algorithm.*/;
    }
else {
    /*Use a real mode algorithm.*/;
    }
```

Real Mode Guidelines

GUIDELINE: Use only those procedures that are available to real mode programs.

Certain operations (such as SgFromSa) are available only in protected mode. These operations are generally not useful to real mode applications and are therefore not available to them. If a real mode program attempts to use one of these procedures, the program will either fail to link or it will encounter an error at run time.

6

Writing Your Application for International Use

This chapter describes the nationalization functions available under CTOS, and some common ways to use them.

Introduction

The Native Language Support (NLS) facilities of CTOS allow you to write applications that accept input and display output in multiple languages. In addition, you can use message routines to store messages in a separate message file. This allows your application to be translated into other languages more easily.

For the purposes of this discussion, *internationalization* means language-independence. Internationalized source code is written in such a way that the resulting program can run in different languages without modifying the executable file itself. *Nationalization* is the process of making external modifications to internationalized software so that the software runs in a specific language.

NLS can be used by an application in two ways. Applications can use the set of NLS tables and routines, which provide commonly used, nationalized functions. Applications also can use the message procedures to create text messages, which then can be removed from the program code and placed in separate *message files*.

The scope of the NLS tables and their associated routines is system-wide. Message files, on the other hand, are applicable on a program-by-program basis. These two concepts are completely independent. As such, they are discussed separately in this section.

Using the NLS Tables and System Calls

Operating systems that support the CTOS/Open standard include a source file called Nls.asm and a macro definition file called Nls.mdf. Nls.asm consists of a collection of tables used to control several different *internationalizable* aspects of software. Included among the tables, for example, are an uppercase to lowercase character translation table, a date/time format table, and a symbols table for numbers and currency. Most vendors ship an Nls.asm file appropriate for the country of purchase, but in some cases the user modifies this file. To nationalize the operating system, the user:

- makes any necessary changes to Nls.asm
- assembles Nls.asm to create Nls.obj
- uses the **Link** command to link Nls.obj with no stack, creating the NLS configuration file, [Sys]<Sys>Nls.sys

When the operating system is bootstrapped, it looks for [Sys]<Sys>Nls.sys. If the file is present, the operating system loads the contents of the file into memory and makes the NLS tables available to applications.

Each of the NLS tables in Nls.sys begins with a two-character signature to ensure table validity. For example, the Keyboard Encoding table begins with the signature, *KE*. The table data follows immediately thereafter. When the operating system loads the NLS tables, it verifies that the operating system table signatures are correct. Other tables can be added as required. (See "Using Alternative NLS Tables," later in this section.)

You can use the NLS tables through a set of routines in the standard object module libraries. These NLS routines return *nationalized* results depending on the contents of the NLS tables. In most cases, the programmer does not need to know the structure of any of the tables. If an application is written using the appropriate NLS routines, proper results are returned to the program for French, if there is a French Nls.sys, German, if there is a German Nls.sys, and so on. If there is no Nls.sys file, all of the routines return the U.S. standard. Therefore, U.S. systems do not need the Nls.sys file.

Table 6-1 lists the NLS tables, and the procedures that reference them. Individual implementations may have other table, as well. Each of these routines is described in detail in the *CTOS/Open Application Programming Interface Specification*.

Table 6–1. NLS Table Numbers

(Page 1 of 2)

Number	Table	Routine(s)
0	Keyboard Encoding	None (used by operating system keyboard process)
1	File System Case	None (used by operating system file system)
2	Lowercase to Uppercase	NlsCase
3	Video Byte Streams	None (used by video byte streams)
4	Uppercase to Lowercase	NlsCase NlsULCMPB
5	Key Cap Legends	GetNlsKeyCapText
6	Date and Time Formats	NlsStdFormatDateTime NlsFormatDateTime
7	Number & Currency Formats	NlsNumberAndCurrency
8	Date Name Translations	GetNlsDateName NlsParseTime
9	Collating Sequence	NlsCollate
10	Character Class	NlsClass
11	Yes or No Strings	NlsYesOrNo NlsYesNoOrBlank
12		Not Used

Table 6–1. NLS Table Numbers
(Page 2 of 2)

Number	Table	Routine(s)
13	Special Characters	NlsSpecialCharacters
14	Keyboard Chords	None (used by operating system)
15		Not Used
16	Multi-byte Escape Sequences	None (used by operating system keyboard process)
17	Operating System Strings	Called by standard library routines.

Using the NLS Procedures

The NLS procedures can be used like any other CTOS library procedure. The NlsYesOrNo routine is shown below as an example. NlsYesOrNo is designed to allow programmers to parse "yes" and "no" strings quickly and consistently. The procedural interface to this routine is as follows:

```
erc = NlsYesOrNo(pNlsTableArea, pSdParam, &fYN);
```

The first parameter for all the NLS routines is pNlsTableArea. This is a pointer to the NLS tables that should be used for this call. Typically, you would use the system NLS tables loaded at boot time from Nls.sys. To do this, just pass a NULL pointer as the value of pNlsTableArea. The NULL pointer tells the operating system to use the tables loaded at boot time.

Alternatively, you could pass a pointer to a different set of NLS tables, loaded by your program. This might be useful if you need to have a single application that works correctly in two or more languages at the same time. However, most applications simply pass NULL for this parameter.

NlsYesOrNo typically is called to get an answer to a prompt from an Executive parameter that ends in a question mark (?), such as [Confirm each?].

The most common way to use this procedure is in conjunction with the RgParam procedure. This is shown as follows:

```
struct {
      char _far *pb;
      Word       cb;
      } sdParam;

CheckErc( RgParam(iParam, jParam, &sdParam) );
erc = NlsYesOrNo(NULL, &sdParam, &fYN);
if( erc != ercOK ) {
     /* Error condition, not yes or no */
```

The procedure first validates the parameters, then sets the flag, fYN. The flag is TRUE (0xFF) if the user entered some string that matched one of the strings in Nls.sys meaning yes. If, however, the user entered something matching one of the strings meaning no (or left the field blank), the flag is FALSE (0x00).

The Executive, for example, uses NlsYesOrNo to examine the answer to the [Confirm each?] parameter to the **Type** command, and for the [Details?] parameter of the **Files** command. The Executive calls the related routine, NlsYesNoOrBlank, to examine the answer to the [Overwrite OK?] parameter to the **Copy** and **Rename** commands as well as other commands.

Using the NLS Tables

The NLS tables can be used in two ways. The simplest and most straightforward way is to use the NLS routines, as described above. Each routine identifies the proper NLS table and provides appropriate defaults. Alternatively, you can obtain a pointer to a particular table and either use it directly or copy the data from the table into your own program for use later.

As an example, consider lowercase to uppercase conversion. If you are writing a program in which you occasionally need to get the uppercase of a particular character, you could code a call as follows:

```
erc = NlsCase(NULL, bCharLower, &bCharUpper, TRUE);
```

The NlsCase procedure does all the work of finding the proper table and indexing that table to find the uppercase of the character. This method generally is preferred because of its simplicity. However, using the routine does require a small amount of overhead in code size and execution speed.

If you have an application in which you will be performing lowercase to uppercase translations many times per second, you may want to use an alternative approach. This approach requires that you build a default lowercase to uppercase translation table into your program.

To internationalize the table, you need to add some initialization code. In the beginning of your program, you need to find out if the associated NLS table was loaded at boot time. If it was, you should copy the contents of that NLS table over the default values in your built-in table. To do this, you must obtain a pointer to the proper table and copy the table to your local data area.

Listing 6-1 shows the code required to accomplish this. See Appendix D for a listing of the entire program.

```
char rgbLowerToUpper[] = { 0, 1, 2, 3, ... , 255};
/* create a 256-byte array, and fill in desired */
/* default translation value for each entry */

main( ) {

Pointer pLowerToUpper;

 /* Get Pointer to LowerToUpper Table (No. 2) */
erc = GetpNlsTable(NULL, 2, &pLowerToUpper);
if(erc == ercOK )
              /* add 2 to bypass the table signature */
   memcpy(&rgbLowerToUpper, (pLowerToUpper+2), 256);
```

Listing 6–1. Loading an NLS Table into an Application

Your program can then get the uppercase equivalent of a character by using the character itself as an index into this table. The code to do this would be written as follows:

```
bCharUpper = rgbLowerToUpper[bCharLower];
```

Since you are just indexing an array, instead of calling a procedure, your code executes many times faster than it would if you used the NlsCase procedure.

Using Alternative NLS Tables

As mentioned above, the operating system validates the NLS tables shown in Table 6-1 at system boot time. If any of the table signatures disagree with the expected signature, an error message is written to the system PLog. You can modify the table list in two ways: you can substitute an alternative set of NLS tables, or you can create an additional NLS tables file.

Whichever method you use to extend the NLS tables, your application (or system service) initialization code should validate that the tables were loaded correctly. Validation should include a call to NlsVerifySignatures. (For details on this routine, see the *CTOS/Open Application Programming Interface Specification*.)

NlsVerifySignatures returns status code 0 ("ErcOK") if all signatures match or status code 13501 ("Bad signature") if one or more of the signatures do not match the expected value(s).

Linking Alternative Tables with Your Program

To use an alternate table set, customize Nls.asm to suit your needs, then assemble it and give the resulting object file a unique name. Next, link that file into your program. Finally, at run time you must pass the address of the tables in your linked-in module as the first parameter to the NLS routines that your program calls.

Note that this method isolates your program from the NLS tables used by the operating system. If the computer on which your program runs has been nationalized in some special way, your application will not have access to that information.

Using Additional NLS Tables

While the standard NLS tables address the most commonly needed nationalized data, they cannot address all possible cases. Some programs use tabular data that must be nationalized, but is not defined in Nls.sys and cannot be treated as a message. If your program has a need for such data, you can create an additional NLS table file that is specific to your program.

To build program-specific NLS tables, create a new assembly file that contains the contents of the additional tables. If there is an existing table in Nls.asm that is similar to the one you want to add, you should use it as a template. Also, be sure to give your new table a unique table number and signature. After you have defined your table, assemble the customized table file and link it, as if it was the Nls.asm file. The result is a file that might be called, for example, "MyProgNLS.sys."

After you have created your program-specific NLS tables file, you need to gain access to it from your program. To do this, you can use the operations OpenNlsFile and GetpNlsTable, as shown in the following example.

```
CheckErc(OpenNlsFile("FooNls.sys", 10, &Password,
                     sPassword, &pMyNlsArea));
CheckErc(NlsVerifySignatures(pMyNlsArea, 1, &wMyTable,
                             &wSignature) );
CheckErc(GetpNlsTable(pMyNlsArea, MYTABLE, &pMyTable));

/* some operation that uses the table */
MyNlsOperation(pMyTable, param1, param2);
```

Using Message Files

There are several ways to use message files. The following sections explain what message files are, and how they are used.

About Message Files

Message files allow you to remove all messages (character strings) from your program and to place them in a separate file called a message file.

Instead of linking the strings into your program, your program retrieves its messages from this external file. Once this is done, you can nationalize the program's message strings simply by editing the message file.

The message file actually exists in two forms: text and binary. The text form is designed to be readable by people and consists of ASCII entries in the form:

<colon> *number* <colon> <delim> *TextString* <delim>[, <delim> *TextString* <delim>]

The delimiter can be any ASCII character. The portion of the syntax in brackets is optional, and can be repeated as many times as you desire. This allows a text string to be broken into any number of substrings. Each substring can use a differerent delimiter, which can be useful if a message contains your delimiter character.

The following is a sample message:

```
:2000: "This is a sample text message."
```

By convention, a text file has the name ProgramNameMsg.txt.

The binary form is the one your program reads. After you create the text file, you must convert it to binary format before your program can use it.

To convert a message text file to a binary file, use the **Create Message File** utility in the Executive. Fill out the command form as follows:

```
Create Message File
  Text file          ProgramNameMsg.txt
  [Message file]     _____
```

By default, the name of the binary file is the same name as the text file except that the .txt extension is replaced with .bin. So, the binary file created above is ProgramNameMsg.bin.

Strategies for Using Messages

There are several issues to consider when deciding on a strategy for using messages with a particular program. A strategy that is correct for one program may be entirely incorrect for another. Among the issues are the following.

- How would the program work if a message is missing?
- How would the program work without a message file?
- How many messages will be used?
- What are the memory constraints?
- How critical is the speed of message retrieval?

One of the first choices you must make is whether or not you will have default messages linked into your program. For example, the workstation initialization program, SignOn.run, must work correctly even if its associated message file, ExecMsg.bin, is missing. So, in the absence of ExecMsg.bin, the SignOn program continues to work correctly in English.

For most application programs, however, the strings should usually be removed from the source and placed in the message file. This saves the amount of memory that the message strings would normally occupy.

A program that uses message files needs only two built-in messages in the native language: "Message file not found" and "Message not found." If the message file cannot be found during initialization, the program generally should call ErrorExitString, and terminate with the "Message file not found" message, also giving the name of the missing message file.

If the message file was opened correctly but the program receives erc 13503 ("ErcMsgNotFound") when it trys to retrieve a message, you may choose to substitute the "Message not found" message for the one intended. In most cases, this probably does not need to be a fatal error condition. Your program should also display the message number and message file name for the message that was not found, so that the problem can be more easily corrected.

Using Messages as Needed

There are two ways to handle message routines. The first approach is designed for application programs that have a large number of messages and that may need to expand macros within messages. Alternatively, a separate set of routines can be used for system services or for any applications that need a very small number of messages. The full message file scheme is discussed first.

Standard Message Routines

The message routines shown below are the basic set provided in the operating system standard library. In addition to the routines described here, there are two other sets which have similar function.

The "Alt" message routines (such as InitAltMsgFile) allow an application to have multiple message files open at a time. The "Erc" message routines (such as InitErcFile) retrieve messages only from the file [Sys]<Sys>ErcMsg.bin, which contains operating system error message text strings. This section describes the basic set, but the information applies to the "Alt" and "Erc" sets as well.

Finally, there is also a fourth set of message routines, which provide minimal functionality. These routines are useful for programs which require only a small number of messages. See "Using a Very Small Number of Messages," later in this chapter, for more information on those routines.

InitMsgFile	Opens a binary message file for subsequent retrieval of numbered messages.
GetMsg	Retrieves a message from the message file, expands any embedded macros, and places the expanded message in memory.
GetMsgUnexpanded	Retrieves a message from the message file and places the unexpanded message in memory. The length of the unexpanded message can be retrieved using the GetMsgUnexpandedLength procedure.
ExpandLocalMsg	Retrieves a message from memory and either sends the expanded message to the video (frame 0) or copies the message to a user-supplied byte stream.
PrintMsg	Retrieves a message from the message file and either sends the expanded message to video (frame 0) or copies it to a user-supplied byte stream.
CloseMsgFile	Closes an open message file.

(For details on each of these routines, see the *CTOS/Open Application Programming Interface Specification*.)

A program calls InitMsgFile during initialization to establish the message file to use. Then, when it needs to use a message from the message file, it calls GetMsg, ExpandLocalMsg, or PrintMsg to expand and display the message.

Typically, you would create a procedure, such as InitMsgEnvironment, to handle the details of initializing the message file. This routine can be written as shown in Listing 6-2. See Appendix D for a listing of the program that calls this procedure.

The message buffer in the listing, rgbBuffer, is used by GetMsg, GetMsgUnexpanded, and/or PrintMsg when they read from the message file. The size of this buffer should be determined according to the size of the binary message file. Increasing the size of the buffer can increase performance when your message file is large.

```
InitMsgEnvironment(Pointer pbMsgFileName, Word
                   cbMsgFileName, Pointer pBuf, Word
                   sBuf, Pointer pCache, Word sCache)
[
ErcType erc;

erc = InitMsgFile(pbMsgFileName, cbMsgFileName, 0, 0,
                  pBuf, sBuf, pCache, sCache);
if(erc != ercOK)
    ErrorExitString(erc, &rgbCantAccessMsgFile,
                    strlen(rgbCantAccessMsgFile) );
return(erc);
}

main()
{
Word    rgbBuffer[0x200];
Word    rgbCache[0x400];
char    MsgFileName[] = "ProgMsg.bin";

InitMsgEnvironment(MsgFileName, strlen(MsgFileName),
                   rgbBuffer,sizeof(rgbBuffer), rgbCache,
                   sizeof(rgbCache));
```

Listing 6–2. Setting Up the Message File Environment

The cache buffer, rgbCache, is used to keep a cache of the most-recently-used messages. The size of the cache buffer should be determined by how often you reuse messages. If messages are never reused, you should use the minimum of 1024 bytes.

Message File Macros

Each message may have one or more macros embedded in the text. These macros are expanded when the message is displayed, making message creation more flexible. A macro is identified by a leading percent sign (%), followed by one or more characters with no spaces. The meanings of these characters are described below.

Macro	String to be insterted
%U	Workstation user name (which expands to NULL if the user name in the Application System Control Block is of 0 length)
%Dn	Date/time formatted with template number n
%Kn	Key cap text for key number n
%nD	Display nth parameter as date/time
%nN	Display nth parameter as a number
%nS	Display nth parameter as a string
%n	Display nth parameter as a string (the default)

As an example, assume the following message:

```
:2000: "Copying on %D10 from %0S to %1S ..."
```

Further assume that the date was Jan 1, 1989 and the strings *Abc* and *Def* were supplied. (For details on the different date time expansions, see Appendix B.) The expanded message would be

```
Copying on 01/01/89 from Abc to Def ...
```

The Executive was one of the first programs to use message files. You can look at [Sys]<Sys>ExecMsg.txt for examples of how to construct messages.

Following is a detailed example of how you would write code to display such a message with macro expansion.

Assume you create the message

```
:3000: "There are %0N files in directory named %1S."
```

This message takes two parameters: the first is numeric and the second is a string. The parameters are passed in an array of sdType parameters. An *sdType* is a 6 byte block of memory in which the first 4 bytes contain the address of the parameter, and the last 2 bytes contain the parameter size. The following code would be used to output the full message:

NOTE: The size passed in the sdType parameter for numeric parameters determines how the number macro is expanded. A size of 1 implies a byte value; 2, a word value; and 4, a double word or quad value.

```
main( )
[
sdType rgSd[2];
Word   iMsgNum;
Word   nFiles;
Word   cbDirName;
char   rgbDirName[12];
.
.
.
rgSd[0].pb = &nFiles;
rgSd[0].cb = 2; /* two bytes in a word */
rgSd[1].pb = &rgbDirName;
rgSd[1].cb = cbDirName;

CheckErc(PrintMsg(iMsgNum, &rgSd, 12
/* two six-byte sdParams */, &bsVid);
```

Listing 6–3. Displaying a Message from a Message File

Pre-Loading Messages

As discussed above, most application programs load messages on an as-needed basis. In some cases, though, pre-loading is the better method.

You should preload the text messages if your program requires the fastest possible performance. If your prime consideration is memory usage, however, you should only load messages as needed. Sometimes, you may want to use a combination of the two.

Preloading messages requires your program to get its messages from the message file and load them into memory during initialization. To do this, you create an array of message numbers for the program to use. Then, during initialization, your program calls GetMsgUnexpanded to load a copy of each message into memory for quick access later. (Since these messages may have macros, you should use GetMsgUnexpanded. That way, you only expand the macro(s) when you actually use the message.) For each message used, you should call AllocAreaSL or ExpandAreaSL to get the necessary dynamic memory in which the message can be stored.

Listing 6-4 shows a code fragment that pre-loads messages into a program's memory. It assumes that InitMsgEnvironment, or a similar procedure, has already been called.

```
sdType  rgMsgArray[MSGMAX];
char    rgbTempMsgBuffer[MAXMSGLENGTH];
.
.
.
i=0;
while(erc == 0 && i < MSGMAX ) {
   erc = GetMsgUnexpanded( (FIRSTMSGNUM+i),
     &rgbTempMsgBuffer,MAXMSGLENGTH,&(rgMsgArray[i].cb));
   if(erc == ercMsgNotFound) {
      WriteByte(&bsVid, NewLine);
      MsgErc(ExpandLocalMsg(&params, sizeof(params),
             &rgbMsgNotFound, strlen(rgbMsgNotFound),
             &bsVid, 1024, &i, fTRUE) );
      break;
      }
   else CheckErc(erc);

   if(i == 0)
      CheckErc(AllocAreaSL( (rgMsgArray[i].cb+1),
                           &(rgMsgArray[i].pb)) );
                                          continued...
```

```
else {
   selectorof(rgMsgArray[i].pb) =
               selectorof(rgMsgArray[i-1].pb);
   CheckErc(ExpandAreaSL( (rgMsgArray[i].cb+1),
            selectorof(rgMsgArray[i].pb),
            &(offsetof(rgMsgArray[i].pb)) ) );
   }

memcpy(rgMsgArray[i].pb, rgbTempMsgBuffer,
       rgMsgArray[i].cb);
i++;
} /* end of while */
```

Listing 6–4. Pre-Loading Messages into Program Memory

If your application has linked-in text strings and you want to overwrite them by pre-loading messages, the preloading should follow the process below.

- For each message, call GetMsgUnexpanded to retrieve the message from the message file and to place the message in a temporary buffer.
- If the size of the loaded message is less than or equal to that of the linked-in message, you can overwrite the linked-in message. (Remember to overwrite the size also.)

You may also decide to use some combination of the above methods. For example, you could preload messages that are always used and choose to load all other messages on an as-needed basis.

Using a Very Small Number of Messages

As mentioned earlier, there is an alternative set of routines that can be used for system services or other programs that use a minimal number of messages. This approach allows your program to copy the entire contents of the message file into a memory buffer and then extract the messages from there. These routines do not support any macro expansion. The three routines listed below can be called.

OpenServerMsgFile	Opens a message file.
GetServerMsg	Retrieves a message from the previously opened message file.
CloseServerMsgFile	Closes the message file.

For details of each of these routines, see the *CTOS/Open Application Programming Interface Specification*.

OpenServerMsgFile performs the following operations:

- opens the specified message file
- calls GetFileStatus to determine the size of the file
- allocates enough memory to contain the message file
- reads the entire contents of the file into memory

To retrieve a particular message, your program must call GetServerMsg. When your program is finished using messages, it should call CloseServerMsgFile, which closes the message file and deallocates the memory it allocated for messages.

Listing 6-5 shows a sample use of the server message file routines. A program that uses the routine can be found in Appendix D.

```
InitServerMsgEnvironment(Pointer pbMsgFileName,
                         Word cbMsgFileName)
{
ErcType erc;

erc = OpenServerMsgFile(pbMsgFileName, cbMsgFileName, 0,
                        0, FALSE);
if(erc != ercOK)
    ErrorExitString(erc, &rgbCantAccessMsgFile,
                    strlen(rgbCantAccessMsgFile) );
return(erc);
}
```

continued...

```
main()
{
sdTypesdMsgRet;
Word    iMsgNum = 56100;
Word    cbRet;
char    rgMsgFileName[] = "ServerMsg.bin";

InitServerMsgEnvironment(&rgMsgFileName,
                         strlen(rgMsgFileName) );

MsgErc(GetServerMsg(iMsgNum, &sdMsgRet) );

WriteByte(&bsVid, NewLine);
WriteBsRecord(&bsVid, sdMsgRet.pb, sdMsgRet.cb, &cbRet);
```

Listing 6–5. Using the Server Message File Procedures

7

Tips for the Application Writer

Introduction

This chapter describes some common practices which we recommend following. Following these practices promotes a common user interface, and enhances usability of your application and of the system as a whole.

Program Structure and Design

CTOS programs written in a high-level language (for example, in C) or in Assembly language can be viewed at three different levels.

- The program level
- The module level
- The procedure level

The program level is the highest-level view. It consists of the executable image, or run file. When the program is started, this file is loaded into memory from disk. It then performs the work it was designed to do. Each program is made up of one or more modules.

A module is an individual source code file, which has been compiled or assembled to object form. The code in each module should perform related functions. Large programs should be built of multiple, logically grouped modules. Programs written in this manner are much easier to maintain than those which use a single, monolithic source file. Once the modules for a program have been created, they can be linked together to form the executable program.

Each module contains one or more procedures. A procedure is simply a section of code which can be called by a symbolic name. This allows commonly-used functions to be written only once, but used at multiple locations in a program. The principle of modularity used for modules applies to procedures, as well. Multiple, specific procedures are usually more efficient than a single, general-purpose procedure with many options.

The source code in each module is compiled or assembled into a relocatable object module. As mentioned above, these object modules can then be linked together by the Linker into a run file. Object modules can also be stored in object module libraries which group many individual object modules in a common repository. The Linker extracts the individual object modules from the library at link time, as needed.

The Linker makes two passes when linking object modules into a program. During the first pass, the Linker reads each module and extracts symbol information from them. It puts this information in a symbol table. The Linker then searches the available libraries for occurrences of those symbols, and extracts the appropriate object module (or modules) from the libraries. Both CTOS.Lib and CTOSToolKit.lib are searched automatically by default during this process.

On the second pass the Linker inserts relocation information in the program image, so that the symbols in the object modules are replaced by their offsets in the program image. Finally, the Linker writes the newly-created executable run file to disk.

The Linker can create any of several different run file types, depending on the command used, and on the options specified. Portable programs should use the Bind command, not the Link command. Portable programs should also use only the following entries for the Run File Mode parameter.

- Real

  This entry creates a real mode run file. This type of run file executes in real mode only. It therefore runs only in the lowest one megabyte of memory on a workstation. This type of run file cannot access extended memory.

- Protected

  This entry creates a protected mode run file, with addressing based on a Local Descriptor Table. This type of run file executes in protected mode on systems that support protected mode, and in real mode on systems that do not. Protected mode run files can be located anywhere in memory. This is the most commonly used run file mode for portable programs.

- GDTProtected

  This entry creates a protected mode run file, with addressing based on the system's Global Descriptor Table. As with the Protected entry, this run file executes in protected mode on systems that support protected mode, and in real mode on systems that do not. This entry allows other programs to gain direct access to this program's code and data. This option is used almost exclusively by system-common services.

Keyboard-Handling Conventions

Application-Independent Key Meanings and Their Use

Certain keys and combinations of keys should be handled by any application. Some are taken care of by the operating system (for example, **Action-Finish**), but others are the responsibility of the application. Table 7-1 describes the common key usage convention for CTOS applications.

Table 7-1. Application-Independent Key Meanings

(Page 1 of 2)

Key Sequence	Usage
Action-Finish	Application takes no action, unless it has disabled Action-Finish. Operating System terminates application.

Table 7-1. Application-Independent Key Meanings
(Page 2 of 2)

Key Sequence	Usage
Cancel	Stop current operation, and return to previous prompt, menu, or state.
Code-"feature key"	Perform the application-defined operation specified by the Code-"feature key" combination.
F1 to F10	Display menu, or prepare to perform an operation. Wait until the Go key is pressed to perform the operation.
Finish	Display exit prompt, and wait for the Go key to be pressed.
Go	Perform the requested action.
Help	Display help information.
Next	Move to the next text field.
Shift-Code-"feature key"	Perform the application-defined operation specified by the Shift-Code-"feature key" combination.

Application-Dependent Key Use Conventions

Certain keys and key combinations may not be needed by all applications, but should perform a consistent function when they are used. For example, not all applications need to implement screen scrolling, but those that do should use the scrolling keys in a consistent manner.

Table 7-2 describes the usage conventions for these keys.

Table 7-2. Application-Dependent Key Meanings

(Page 1 of 2)

Key Sequence	Usage
Arrow Key	Move the cursor one character position (or one line) in the direction of the arrow.
Bound	Highlight all characters between the current cursor position and a previously-marked cursor position. Prepare to perform an editing function on the marked block of text.
Code-Arrow Key	Move the cursor as far as it can go in the direction of the arrow. For example, to the end of the current line or to the top of the current video frame.
Code-Mark	Unmark a previously-marked block of text.
Code-Next Page	Insert New Page character (0Ch).
Code-Scroll Up/ Code-Scroll Down	Scroll the contents of the screen up/ down one full screen.
Copy	Copy a previously-marked block of text to the current cursor location. Displaces, but does not destroy existing text.
Delete	Delete the single character or the marked block of text at the current cursor location.
Mark	Highlight the character at the current cursor position, and prepare to perform an editing function on it.
Move	Move a previously-marked block of text to the current cursor location. Displaces, but does not destroy existing text.

Table 7-2. Application-Dependent Key Meanings
(Page 2 of 2)

Key Sequence	Usage
Next Page/ Prev Page	Scroll the contents of the screen up/ down one full page or one full screen.
Scroll Up/ Scroll Down	Scroll the contents of the screen up/ down one line.
Shift-Arrow Key	Move the cursor five character positions (or five lines) in the direction of the arrow.
Shift-Scroll Up/ Shift-Scroll Down	Scroll the contents of the screen up/ down five lines.

Keyboard Events an Application Must Handle

An application should make a reasonable response to any input it receives from the keyboard. If the input is not what the application expects, it should beep. It should also display an error or prompt message, if one is not already displayed.

Screen Layout Conventions

While no strict guidelines currently exist for an application's screen layout, certain conventions have been widely used historically. Following these conventions promotes a consistent look between applications.

Function Key Menus

If an application uses function keys, it should provide a function key menu on the screen. The function key menu should appear at the bottom of the screen, so that it is easy to relate the menu to the function keys on the keyboard. If an application needs to use the entire screen, it should display its function key menu when the Help key is pressed.

For an example showing how to display a function key menu on the screen, see "Video Management – VAM and VDM" in Chapter 2.

Help

Applications should provide some form of help to the user. At the least, the application should display a list of commands or function key meanings when the user presses the Help key.

Listing 7-1 shows a VAM procedure which displays a help window when called. The procedure sets up its window, displays a message, then prompts the user for a response. In this case, the only accepted response is Cancel, which causes the routine to restore the previous contents of the screen and then return to the caller.

```
/* display a help message */
void PutHelp() {
char    *pHelpMsg = "You Pressed the Help key.  Press
Cancel to continue.";
Word    sHelpMsg = 51;
Word    sFrameBuf;
char    inchar;

/* frame buffer is number of chars in frame */
sFrameBuf = 3 * Cols;

CheckErc(InitVidFrame(HELPFRAME,0,Lines-6,Cols, 3,0,0,0,
          0,0));
CheckErc(QueryFrameCharsAndAttrs(HELPFRAME, 0, 0,
          pbTextBuf, sFrameBuf, pbAttrBuf, sFrameBuf));
CheckErc(ResetFrame(HELPFRAME));
CheckErc(PutFrameChars(HELPFRAME, 0, 0, thinline, Cols));
CheckErc(PutFrameChars(HELPFRAME,0,2,pHelpMsg,sHelpMsg));

CheckErc(ReadKbd(&inchar) );
while(inchar != Cancel) {
    Beep();
    CheckErc(ReadKbd(&inchar) );
    }

CheckErc( PutFrameCharsAndAttrs(HELPFRAME, 0, 0,
          pbTextBuf, sFrameBuf, pbAttrBuf, sFrameBuf) );
}
```

Listing 7–1. Sample Help Window Display Procedure

Status Information

Status information that applies to an entire program, such as the time of day, should be displayed at the top of the screen. Status information that applies to a particular video frame, such as a filename in an editor, should be displayed at the top of that video frame.

Creating An Executive Screen

Many utilities use the same screen display as the Executive for their user interface. Such utilities can use the CreateExecScreen procedure to set up the screen, rather than implementing the setup code themselves.

The CreateExecScreen procedure takes three parameters: the size of the video character map, the number of columns on the screen, and the number of lines on the screen. These parameters should NOT be hard-coded in the program, but should instead be retrieved by the QueryVidHdw procedure. Using QueryVidHdw allows the program to work with displays that have varying numbers of rows and columns.

Listing 7-2 shows a code fragment that sets up an Executive-type screen display using CreateExecScreen.

```
/* first find out the screen dimensions */
CheckErc(QueryVidHdw(&rgbVidHdw, sizeof(rgbVidHdw) ));
nLines = rgbVidHdw[1];
nCols = rgbVidHdw[2];

/*now reset the video and get the size of the char map */
CheckErc(ResetVideo(nCols, nLines, TRUE, BLANK, &sMap));

/* now create the executive screen */
CheckErc(CreateExecScreen(sMap, nCols, nLines));
```

Listing 7–2. Creating an Executive Screen

The Executive screen consists of three display frames, numbered 0 through 2. Each is described below.

- Frame 0 – the user input frame

  This is the primary frame, and consists of the entire screen except for the top three lines. All user input and user output should be displayed in this frame.

- Frame 1 – the status frame

  This frame consists of the top two lines of the screen. This frame should display status information, such as the current path and the current date and time.

- Frame 2 – the event frame

  This frame consists of the third line of the screen, and generally contains a horizontal line separating the status frame from the user input frame. This frame is used to display system events, such as mail notification.

After the call to CreateExecScreen, these frames are initialized, but empty. It is the application's responsibility to fill in the frames with the appropriate information.

To be consistent with other applications that use the Executive screen, the application should display its name in the upper left-hand corner of the status frame. It should also display the current date and time in the lower right-hand corner of the status frame.

If an application requires a customized video display, see "Video Management – VAM and VDM" in Chapter 2, "Basic Input and Output."

Cleanliness

Programs should observe general programming cleanliness guidelines, so that future enhancements to the operating system do not break them.

Guidelines for Screen Handling

In most implementations of CTOS, it is possible to get the address of the memory in which the screen image is stored, and to write directly to video memory. It is common practice in some operating systems, such as MS-DOS, for applications to write directly to video memory.

Portable CTOS applications should never write directly to video memory. They should instead use the VAM and VDM calls to display information on the video screen. This allows compliant applications to continue executing in the background when they are used with a context management shell, such as the Context Manager. Applications that write directly to the screen execute only when they are the foreground process.

File Suffix Conventions

Applications that have special-purpose files should append a descriptive suffix to those files. Table 7-3 describes some filename suffixes commonly used by application programs for their files.

Table 7-3. Common File Suffixes

(Page 1 of 2)

Filename Suffix	Usage
.bin	A binary message file.
.config	A configuration file.
.fls	A file that contains a list of file specifications.
.lib	A library of object module procedures.
.lst	A listing file.
-Old	The original version of a file that is currently being modified.
.sub	A submit file.
.sys	A system file.
.tmp	A temporary file created by an application.

Table 7-3. Common File Suffixes
(Page 2 of 2)

Filename Suffix	Usage
.ts	A typescript file, used for error recovery.
.txt	A text file.

The Scratch Volume

All CTOS systems have a special volume, the scratch volume, for the storage of temporary files. The volume name of the scratch volume is [Scr], and it behaves like any other CTOS volume. Application programs can create and delete files and directories on the scratch volume at will.

In many installations, the scratch volume is mapped to the user's [Sys] volume. However, the scratch volume could just as well be mapped to a RAM disk, for performance enhancement. Therefore, only files which are truly temporary should be stored on the scratch volume. In addition, temporary files on the scratch volume should be deleted after the application is finished using them.

Any file that may be needed for recovery after a system failure *must not* be stored on the scratch volume. Such files should be stored as temporary files on the [Sys] volume or on some other real volume.

8

Writing Request-Based System Services

Introduction

Two types of system services exist: request-based system services, and system common services. The two are somewhat similar, in that each performs a service for a client application, but the methods by which the two types perform the service are quite different.

Request-based system services are the most common type. A request-based system service exchanges messages with its clients, in order to perform work for them. The client and the system service are completely separate processes within the machine, which exchange data and cooperate in order to perform their work.

A request-based system service can reside anywhere in a network. It need not be on the same processor as its client.

System common services are less widely used, mainly because they are a fairly recent feature of the operating system. A system common procedure is a globally accessible procedure, managed by the operating system. When a client calls a system common procedure, that procedure executes as if it were part of the client application.

A system common procedure must reside on the same processor as its client. For more information on system common procedures, see Chapter 9, "Writing System-Common Services."

Requests and Request Levels

Request-based system services communicate with their clients using a request-response model. In the abstract model, the client sends a request to the service, and the service responds to it. In reality, the client builds

a data structure called a request block, which contains everything the service needs to process that request. The service receives the request block, performs the requested work, then returns the request block to the client.

Table 8-1 shows the format of a sample request block.

Table 8-1. A Sample Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	1	Size of Control Info
1	RtCode	1	Routing Code
2	nReqPbCb	1	# of Request Data Items
3	nRespPbCb	1	# of Response Data Items
4	userNum	2	Client's User Number
6	exchResp	2	Response Exchange
8	ercRet	2	Status Code Returned
10	rqCode	2	Request Code
12	fh	2	File Handle (control info)
14	lfa	4	Logical File Address (control info)
18	pbData	4	Pointer and Size of Data
22	cbData	2	(request data item)
24	pbCountRet	4	Returned Byte Count
28	cbCountRet	2	(response data item)

The request block contains several kinds of information. It contains information that describes itself, so that the operating system can pass it to the system service. It contains a request code, so that the operating system can identify the desired system service. Third, it contains two types of application-specific data.

The application-specific data in a request block can be request data, or response data. The application uses request data to pass information to the system service. When the system service has finished processing the request, it uses the response data areas to pass information back to the client application.

Request data should always appear before response data in the request block. The operating system expects response data items to be located at the highest offsets in the request block.

Every request has a request code. The request code is simply a number that uniquely identifies that type of request. Each system service usually serves several request codes, including codes for data requests from clients, for special operating system requests, and for a deinstallation request.

Since each request code must be unique, only one system service can use a given request code. If two system services try to use the same request code, one or both of the services will fail. For this reason, Unisys allows developers to register a certain range of request codes. Once you have registered a request code, it is yours alone. No one else should use it. You can reserve request codes in Level A or B. Registering request codes is discussed below in "Registering Request Codes."

Table 8-2 shows the categories of request codes, and their numerical ranges. Levels 0 through B are reserved for internal and for registered use. Levels C through F are available for unrestricted use.

Table 8-2. Request Code Levels

(Page 1 of 2)

Level	Hexadecimal Values
0	0000 to 0FDF and FFE0 to FFFF
1	1000 to 1FFF
2	2000 to 2FFF
3	3000 to 3FFF
4	4000 to 4FFF

Table 8-2. Request Code Levels
(Page 2 of 2)

Level	Hexadecimal Values
5	5000 to 5FFF
6	6000 to 6FFF
7	7000 to 7FFF
8	8000 to 8FFF
9	9000 to 9FFF
A*	A000 to AFFF
B	B000 to BFFF
C*	C000 to CFFF
D	D000 to DFFF
E*	E000 to EFFF
F	F000 to FFDF

* User levels which support a procedural interface.

Registering Request Codes

There are two ways to obtain registered request codes in the A and B ranges; by FAX or by telephone.

When you register request codes, you ask for a certain quantity of them, and Technical Support assigns the numbers for you.

By Telephone

To register request codes by telephone, call the Unisys Network Computing Group in San Jose, California, USA. The telephone number is:

(408) 434-2848

You must call between 9AM and 5PM United States Pacific Standard Time (or Pacific Daylight Time in the Summer months).

Ask for the CTOS/BTOS Technical Support Department.

By FAX

To register request codes by FAX, send your FAX to the following United States telephone number:

(408) 434-2131

Address your FAX to "CTOS/BTOS Request Codes." If you fail to address your FAX to "CTOS/BTOS Request Codes" it may be misrouted. The FAX number is available 24 hours a day.

All FAXed requests to register request codes *must* include the following information:

- Your name. This consists of your company name and the name of a person who can be contacted if questions arise about your request. You should also include a telephone number and a FAX number where the contact person can be reached.
- The total number of request codes WITH a procedural interface that you need.
- The total number of request codes WITHOUT a procedural interface that you need.
- A full description of the request block for each request you want to register. See Table 8-1 for the format of a request block.

  Do not fill in the rqCode field.

Technical Support will notify you which request code numbers they have assigned to you as soon as possible after receiving your FAX.

Conserving Request Codes and Memory

Request codes are a finite resource. They should not be wasted. In addition, efficient use of request codes has a direct relation to the size of the operating system.

Conserving Memory with Efficient Request Code Use

CTOS maintains a table in memory for each request code level. This table contains entries for all requests in that level, up to the number of the highest one currently defined. Even if a request code number is undefined, CTOS reserves a slot for it if any request code with a larger number *is* defined.

For example, assume that on a particular workstation the highest currently-defined request code in level E is 0E187h, and you add a new request code as number 0EF00h. The operating system must now reserve table entries for all the request codes between 0E187h and 0EF00h. Each table entry requires 8 bytes, which in this case adds up to a lot of wasted memory.

This is one reason Technical Support assigns request codes in the A and B levels. By assigning request codes incrementally, they can keep the number of wasted request table slots to a minimum.

Conserving Request Code Numbers

Use the following techniques to keep the number of request codes needed by your system services to a minimum:

1. When several service functions require similar parameters, add a "CaseValue" field and use a single request code for all of them.

2. Use the same request for Termination, Abort and Swapping requests. The three can be distinguished by the contents of the request block. For more information on these requests, see "Termination and Abort Request" and "Swapping Requests" later in this chapter.

Efficient use of request codes may not be necessary if you are writing a system service purely for your own use. However, if you ever plan to distribute it or to sell it, both your customers and the operating system developers will appreciate your efforts.

Operations Performed by a System Service for Applications

A request-based system service performs functions for its client programs. The function a system service performs can be anything at all, but most often a system service controls access to a shared resource. For example, the queue manager is a system service.

Server/Client Communication

The service and its client communicate using two kernel primitives, Request and Respond. The client uses the Request primitive to send a request block to the system service. The service processes the request, then uses the Respond primitive to send the same request block back to the client. The service sends the request block to the exchange specified by the client in the response exchange field of the request block.

The client can issue the Request primitive directly, or it can use a request procedural interface for the request. If the client uses the Request primitive, the client must build the request block itself. If the client uses the request procedural interface for a request, the operating system builds the request block for the client. Most clients use the request procedural interface.

When the operating system detects that a Request primitive was issued (either directly or through a procedural interface), the operating system kernel checks the request code in the client's request block. It then uses the request code as an index into its request routing table, which contains the exchange number of the service that serves each valid request code. The operating system looks up the exchange at which the request is served, then sends the address of the request block to that exchange.

When the system service receives the request block's address, it first checks the contents of the request block. If the information is valid, the service performs the requested service for the client, and fills in any response information in the request block. In addition, the service enters a value of zero (ercOK) in the status code field of the request block. If the service encounters an error, or if the request block is not valid, the service enters the appropriate error code in the status code field of the request block.

Finally, the service responds to the client by calling the Respond primitive. The operating system then routes the response back to the client's exchange, where the client has been waiting for it.

The Server/Client Relationship

When a server provides a service to a client, the two establish a relationship between them. This relationship may last only for the time it takes to respond to a single request, or it may last for a longer term. Much of the discussion in the following sections treats events differently depending on how they make use of this client/server relationship.

Most of the time, the client establishes a long-term relationship with the server, so that the two can exchange a series of requests and responses. A typical example of this is reading a sequential file. All the read operations affect the same file, and each one depends on the successful completion of all the previous ones. In such cases, the service must remember not only which file is being read, but also who the client is, in order to close that file if the client terminates abnormally.

This kind of long-term relationship is called a *connection*. When a client needs to use a service over a period of time, the client must first establish a connection, then use that connection in a series of requests. Then, when it is finished, it must destroy the connection. Any one connection links one client program to one system service. Clients may have several simultaneous connections to the same or to different services, and vice versa. Each connection, however, is independent of all other connections.

Connections are not peer-to-peer. The client always initiates each communication, and the service replies to it. These roles apply to only one particular connection, however. A given program may act as the server on a connection to one program and as the client on a connection to another program.

Connection Establishment

To establish a connection, the client submits a request to the server. This request must identify which system service the client wants to use. This can be done explicitly or by default. In most cases, the client also

identifies a specific resource that it wishes to access. The identification of the resource can sometimes also be used to identify the server. This is the case for the file system, for example.

Assuming that the resource is available, and that the client has the right to access it, the server establishes the session by returning a reference number, which is used to identify all requests that pertain to this session. This reference number is called a *handle*. All connection handles are treated as if they were file handles, making them easier to work with. If the resource is unavailable, or if access to it is barred, the server returns an error code instead of a handle.

The handle is used not only by the client and the server, but also by the operating system and the communications network to refer to the connection. To ensure that all handles are unique, the network, the operating system, or both may need to map the handle to a different value. Neither the client nor the server should assume that the other sees the same value for the handle as it does. The only guarantee is that the handle on all subsequent requests over a connection will have the same value as the one returned when the connection was opened.

Listing 8-1 shows how a system service might handle a connection opening request.

```
/* Open a connection, return a handle */
void OpenConnection(OpenRqType *pRq)
{
Word wHandle;

/* find an available handle */
wHandle = 0;
while( (rgHandles[wHandle] != 0) &&
       (wHandle < MAXCONNECTIONS) )
    wHandle++;

/* if all the handles are in use */
if(wHandle == MAXCONNECTIONS) {
    pRq->RqHead.ercRet = ercNoHandles;
    }

/* if one is available */
else {
    /* put the handle number in returned handle field */
    *(pRq->pHandleRet) = wHandle;
```

continued...

```
    /* put client's usernum in the handle array at */
    /*  position wHandle */
    rgHandles[wHandle] = pRq->RqHead.userNum;
    /* return ercOK and increment connection count */
    pRq->RqHead.ercRet = 0;
    cOpenConnections++;
    }
Respond(pRq);
}
```

Listing 8–1. An Open Connection Request

A Note About Handles

As mentioned above, all handles are treated as file handles, which are 16-bit numbers. The top four bits of the handle are used by the operating system for routing, leaving 12 bits for the service to use. The meaning of the top four bits (bits 12 to 15) are listed below.

- Bit 12. Unused. Should always be 0.
- Bits 13 and 14. Can be set by the system service to indicate that the service resides on the master.
- Bit 15. Set by network software to indicate that the service resides on a remote network node.

Most system services can simply set all four bits to zero, and let the operating system take care of the routing. However, there is one special case for cluster routing.

If the same system service may be installed on both a cluster workstation and the master, the operating system on the cluster workstation must be able to identify which handles belong to the service on the master.

To accomplish this, the system service on the master should set bits 13 and/or 14 in its handles, and the service on the cluster workstation should leave them both unset. The following two code fragments show how a service can determine whether it resides on a master workstation, then set the appropriate bits.

The first determines whether a program resides on the master, and sets a flag accordingly. The second sets the appropriate bits in the handle.

```
/* Get the system config. block structure */
CheckErc(GetpStructure( 0x2C8 ,0, &pMySysConfigRet));
/* check byte at offset 33, clusterconfig byte */
if(pMySysConfigRet->ClusterConfiguration == 2)
    fMasterOS = TRUE;
else
    fMasterOS = FALSE;


if(fMasterOs == TRUE)
/* OR the handle with 0x6000 to set bits 13 and 14 */
    wHandle = wHandle | 0x6000;
/* put the handle number in returned handle field */
*(pRq->pHandleRet) = wHandle;
```

The Real Work

Once the connection is established, the client and server use requests to manipulate the resource that was allocated for the connection. Typically these requests perform read, write and/or control operations. The requests are guaranteed to be delivered to the server in the order in which they were sent. The server can respond to them in any order, however.

Connection-oriented requests do not identify the server or the resource they affect. They must specify the connection handle instead.

If a request is routed to a remote processor, it cannot exceed 2.5K bytes in size. If a request, such as a disk read, requires more than 2.5K of returned data, the operating system takes special action. The client still issues only a single request, but the operating system breaks it up. The result is that the server receives several separate requests, each for less than 2.5K of data. Then, the operating system collects the server's responses into a single response to match the client's single request.

The only difficulty arises if errors occur on more than one of the server's requests. The server can return one error code on each, but the client can only receive one error code, because it only issued one request. If this happens, the operating system returns the numerically highest error code. For this reason, more severe error codes should be assigned higher numbers.

Listing 8-2 shows a function that returns a string of data in response to a request. The client passes a valid connection handle, the address and size of a buffer, and the address of a variable in which the server returns the actual size of the returned string.

```
void ProcessDataRequest(DataRqType *pRq)
{
Word    i=0;

/* check for valid handle */
if(pRq->RqHead.userNum != rgHandles[(pRq->wHandle)] ) {
    pRq->RqHead.ercRet = ercInvalidHandle;
    TerminateIfErcNotOk(Respond(pRq), (RqHeaderType *)
                            pRq);
    return;
    }
/* calculate the maximum string size */
if(pRq->cbDataRet < srgMsgRet)
    i = pRq->cbDataRet;
else
    i = srgMsgRet;

/* put the message and the size in the user's buffer */
memcpy(pRq->pbDataRet, &rgMsgRet, i);
*((Word *)(pRq->psDataRet)) = i;

/* respond to the request */
pRq->RqHead.ercRet = 0;
TerminateIfErcNotOk(Respond(pRq), (RqHeaderType *) pRq);
}
```

Listing 8–2. Processing a Data Request

Connection Termination

When a client is finished with a connection, it should request the server to terminate the session. This type of request is generally called a *close* request. A close request asks the server to release any resources associated with a given connection handle and to terminate the connection with the client who owns that handle.

When it receives a close request, the server must first reply to any outstanding requests on that connection. It can then reply to the close request and deallocate any resources associated with the connection.

```
/* Close connection procedure.  Assumes that the */
/* service has already served any outstanding requests */
void CloseConnection(CloseRqType *pRq)
[

/* if the user number matches the value in the array */
/* of handles (if the handle is valid) */
if(rgHandles[pRq->wHandle] == pRq->RqHead.userNum) {
    /* reset the handle and decrement the connection */
    /*  count */
    rgHandles[pRq->wHandle] = 0;
    cOpenConnections--;
    pRq->RqHead.ercRet = 0;
    }
/* if the user number doesn't match rgHandles */
else
    pRq->RqHead.ercRet = ercInvalidHandle;
Respond(pRq);
}
```

Listing 8–3. A Close Connection Request

Connectionless Requests

Short term relationships can be handled by connectionless requests. A connectionless request can be used whenever the execution of the request is independent of all previous and future requests. This means that the server does not need to "remember" anything about the client after it has responded to the request. Connectionless requests are used most often for status requests, configuration updates, and deinstallation requests.

Operations Performed by a System Service for the Operating System

While a client and a server participate in a connection, the client's environment may change in ways that the server must know about, even if the client need not know about them. For example, the client may not know if it is swapped to disk temporarily, but the server must know so that it does not write to memory that no longer belongs to its client.

When such an event occurs, the operating system notifies all servers by issuing a specific request and waiting for a response. The following sections describe each of these requests, and the server's action when one is received.

Termination and Abort Requests

Whenever a client application program attempts to terminate, the operating system issues termination requests. These requests contain the user number of the terminating application, and a status code indicating the cause of termination. The operating system does not finish terminating the client application until it receives a response for each of the termination requests it issued.

The operating system issues a termination request when an application terminates for normal reasons, such as successful completion or detection of an error. For example, the application may have called Exit or ErrorExit, or the user may have pressed **Action-Finish**.

The operating system issues an abort request when an abnormal event causes the application to terminate. For example, the master issues an abort request for a client if it loses communication with the cluster workstation on which the client resides.

When a service receives a termination request, it must first respond to all outstanding requests from the terminating client. It can then deallocate any resources owned by the client, and respond to the termination request. A service must always respond to all outstanding requests from the client *before* it responds to the termination request. Note that the server's action when it receives a termination request is almost identical to the one for a close connection request.

An abort request is identical to a termination request, except that the application's termination status code is always 8200. Some services may need a separate procedure to handle abort requests, but most can use the same request for both termination and abort.

Table 8-3 shows the format of a termination request. The userNum field contains the terminating user number. The ercRet field should be set to 0 by the server. The ercTermination field contains the status code with which the application terminated (for example, 0 for normal termination).

Table 8-3. Termination Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	User Number
6	exchResp	2	Response Exchange
8	ercRet	2	Status Code Returned
10	rqCode	2	Termination Request Code
12	ercTermination	2	Termination Status Code

Listing 8-4 shows how a service might handle termination and abort requests.

```
/* Terminate connection procedure.  Assumes that */
/* service has already served any outstanding requests.*/
void ProcessTerminationRequest(SystemRqType *pRq)
{
Word wHandle = 0;

/* if the user has one or more open connections */
while(wHandle < MAXCONNECTIONS) {
    if(rgHandles[wHandle] == pRq->RqHead.userNum) {
        /* reset them and decrement connection count */
        rgHandles[wHandle] = 0;
        cOpenConnections--;
        }
    wHandle++;
    }
/* respond with erc 0 */
pRq->RqHead.ercRet = 0;
Respond(pRq);
}
```

Listing 8–4. An Application Termination Request

Swapping Requests

The operating system issues a swapping request when a user application is about to be swapped to disk. The swapping request is used to prevent the service from issuing a response to the user application, until that application has been swapped back into memory.

If a server receives a swapping request for a given user number, and has no outstanding requests from that user number, the service should respond immediately. The service has no need to take special action, even if it has a connection with that user.

If a server has outstanding requests from the user, though, it must handle the swapping request in one of three ways:

- Delay its response to the swapping request until it has responded to all pending requests for the user. This option should only be used if the service can respond to all the pending requests in a small amount of time.

- Reply to all the pending requests with a status code of 37, which indicates "Service not completed." This tells the operating system that those requests have not been processed. The operating system keeps a list of these requests for the swapped user, and reissues them when the user is swapped back into memory. This option should be used only if the user's requests can be suspended indefinitely without causing problems for the server or for its other clients.

- Reply to the swapping request without taking any specific action. This option should be avoided, since it prevents the application from being swapped to disk.

Any server that may hold a request for a significant amount of time before responding to it should serve swapping requests. Failure to do so can randomly prevent applications from being swapped to disk.

Table 8-4 shows the format of a swapping request. The userNum field contains the user number of the user who is being swapped to disk.

Table 8-4. Swapping Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	User Number
6	exchResp	2	Response Exchange
8	ercRet	2	Status Code Returned
10	rqCode	2	Swapping Request Code
12	reserved	2	

Installation and Deinstallation

The life cycle of a system service can be viewed as three consecutive phases. The first is the installation phase, which includes notifying the operating system of the requests it serves, and converting to a system service. Next comes the active phase when the service accepts requests from clients and performs the work it was designed to do. The third phase is deinstallation, which terminates the system service.

Installation

Installing a system service is not particularly difficult, however there are several guidelines that must be followed to prevent undesirable situations.

The critical point in the installation of a system service is the ConvertToSys operation. Certain actions must be performed prior to ConvertToSys, and others must be performed afterward. Performing an action on the "wrong side" of ConvertToSys can result in abnormal termination, or other problems.

A typical installation sequence is:

1. Call GetPartitionHandle to see if it is already installed in another partition. A system service should not install if it is already present on the current processor.

2. Allocate any permanent resources to be used by the system service. This includes allocating exchanges, allocating short-lived memory, and creating any additional processes. No long-lived resources should be allocated before ConvertToSys, however.

3. Query the current status of any requests the service will serve, using QueryRequestInfo. This lets the service make sure the requests are declared in the operating system. If any of its requests are not declared, the service should abort. The service should also save the values returned by the queries so that it can restore them when it is deinstalled.

4. Perform ConvertToSys, then Exit. The ConvertToSys operation changes all processes, exchanges and memory in the partition from application status to system service status. This converts the partition to a system service partition.

   The Exit operation then reloads the exit run file into a new foreground partition, and returns control to the system service program. The Exit operation does not terminate the system service's execution.

5. Change priority, so that the service runs at a higher priority than any potential client application.

6. Use the SetPartitionName operation to identify the service to the operating system.

7. Perform a ServeRq operation for each request the system service will handle. To prevent timing windows, requests from the operating system (such as swapping requests) should be served first, followed by all requests other than connection-opening requests. Connection-opening requests should be served last.

The order of the operations before ConvertToSys can be changed, and the order of the operations after ConvertToSys can be changed. However, compatible programs should generally not move an operation from before to after ConvertToSys, or from after to before.

Listing 8-5 shows an example of how a system service might install itself.

```
void main() {
RqHeaderType   *pRq;
int            i = 0;

    CheckErc(GetUserNumber(&usernum));
    CheckErc(AllocExch(&exchServe));
/* This is the already-installed test.  If already  */
/* present, set up an appropriate exit message, and */
/* exit*/
    if (GetPartitionHandle(&rgServerName, srgServerName,
                        &PhRet) == ercOK){
        ErrorExitString(0, &rgInstMsg1, srgInstMsg1);
    }
/* query the requests. If any are undefined, exit */
    while(i<NUMREQUESTSSERVED) {
        CheckErc( QueryRequestInfo( FIRSTREQUESTNUM+i,
                    &(rqInfo[i]), srqInfo));
        i++;
        }

/* BECOME A SERVER */
/*  Since the server is not already installed, install */
/*  now.  Set up an appropriate exit message, and */
/*  convert to sys (become a server). */
/*  The ErrorExitString call after the convertosys */
/*  call allows the reloading of an exec in the */
/*  primary partition. */

/*  Note: you must do setmsgret before convertosys.  */
/*  Can't print a string to the screen after */
/*  converttosys. */
    CheckErc(SetMsgRet(&rgInstMsg2, srgInstMsg2) );
    erc = ConvertToSys();
    if(erc != 0) {
        ErrorExitString(erc, &rgInstErrorMsg,
                        srgInstErrorMsg);
        }
    else
        Exit(0);

/*  All servers must run at a priority higher than 40, */
/*  else deadlocks can occur. */
    preTerminateIfErcNotOk(ChangePriority(12));

/*  Set the partition name so that the deinstallation */
/*  program can obtain this partition handle.*/
    preTerminateIfErcNotOk(SetPartitionName(0,
                            &rgServerName, srgServerName));
```

continued...

```
/*  Now serve the Request to the service exchange for */
/*  this program. */
    preTerminateIfErcNotOk(ServeRq(SWAPRQ, exchServe));
    preTerminateIfErcNotOk(ServeRq(TERMINATERQ,
                                   exchServe));
    preTerminateIfErcNotOk(ServeRq(DATARQ, exchServe));
    preTerminateIfErcNotOk(ServeRq(DEINSTRQ, exchServe));
    preTerminateIfErcNotOk(ServeRq(CLOSERQ, exchServe));
    preTerminateIfErcNotOk(ServeRq(OPENRQ, exchServe));

/*  Server loop... */
/*      Check for new request.  If there is one, */
/*      process it. */
    while(TRUE) [
        Wait(exchServe, &pRq);
        ProcessRequest(pRq);
        ]
}
```

Listing 8–5. System Service Installation

Deinstallation

System services are trusted entities, and the operating system relies on the fact that they do not disappear unexpectedly. Because of this, terminating a system service requires care.

Usually, the system service writer also writes a utility program that deinstalls the service. Not surprisingly, this program is called a deinstallation utility.

Deinstalling a system service is accomplished in three phases. First, the deinstallation utility sends a request to the system service, telling it to shut down its operation. Second, when the service receives the deinstallation request it must perform the following actions:

1. Check for open connections with client applications. The service should refuse to deinstall if it has any open connections. Note, however, that some application errors could leave an unused connection open. It is the system service's responsibility to handle this possibility (for example, through its abort request handling).

2. Restore the request table in the operating system to its original state, by issuing ServeRq operations with the old exchange numbers the service saved at installation time. These ServeRq operations should be issued in the opposite order as at installation time. In other words, connection-opening requests should be "unserved" first. This ensures that the service receives no new requests.

3. Respond to ALL outstanding requests (except the deinstallation request) with an appropriate error code. This ensures that no requests are lost when the server deinstalls.

4. Close any connections the server may have opened as a client (with the file system, for example).

5. Unlock its own partition by calling SetPartitionLock, then respond to the deinstallation request.

6. Terminate by calling ExitAndRemove.

When the deinstallation utility receives the service's response, it should take appropriate action, based on the status code returned by the service.

Listing 8-6 shows the deinstallation procedure for a sample system service.

```
void DeInstall(pRq)
RqHeaderType   *pRq;
{
RqHeaderType   *pWaitingRq;

/* pRq only points to data when a deinstall request is */
/* received.  Is null when deinstalling on error */
if( pRq != NULL) {
    if(cOpenConnections > 0) {
        /* there are connections open */
        pRq->ercRet = ercOpenConnections;
        Respond(pRq);
        return;
        }
    }
/* if no open connections, or if deinstalling on error */
```

continued...

```
/* unserve the requests */
    while(i<NUMREQUESTSSERVED) {
        CrashIfErcNotOk( ServeRq( FIRSTREQUESTNUM+i,
                          rqInfo[i].exch) );
        i++;
        }

/* Loop here to clear all messages on this exchange */
    while (Check(exchServe, &pWaitingRq) == ercOK) {
        ProcessRequest(pWaitingRq);
            }
/* All done, unlock partition, and deinstall  */
    CrashIfErcNotOk (SetPartitionLock(FALSE));
    Respond(pRq);      /* respond to deinst. rq */
    ExitAndRemove();/* bye */
    Crash(3);      /* bad problem if we ever get here */
}
```

Listing 8–6. System Service Deinstallation Procedure

Deinstallation on Error

If a system service encounters a severe error, it should usually deinstall itself, instead of crashing the system. A system service should only call Crash when the consequences of continuing may be worse than a crash. Listing 8-7 shows a sample procedure that deinstalls a system service in event of an error. The procedure logs an error message in the system log, then deinstalls the service.

```
void TerminateIfErcNotOk(Word erc, RqHeaderType *pRq)
{
    if (erc != ercOK) {
        LogMessage(rgmsgError, strlen(rgmsgError));
        if(pRq != NULL)
            pRq->ercRet = 33; /* service not available */
        DeInstall(pRq);
        }
}
```

Listing 8–7. Deinstalling on Error

System Services that Act as Filters

So far, this chapter has discussed services that simply receive requests and reply to them. However, a service can also intercept a request before it gets to its intended server. A service that intercepts requests is called a filter. Filters can provide a wide variety of functions, usually involving pre-processing or post-processing of requests.

A filter must always be installed on the same processor as the service whose requests it is filtering. The filter must also be installed after that service. Once installed, the filter serves some or all of the requests intended for the filtered service.

A filter does not need to filter all of the requests of the original service, but it usually needs to serve the requests from the operating system (termination, abort, and swapping), and the deinstallation request. The filter must serve the filtered service's deinstall request, so that it can deinstall before the filtered service does. A filter should also intercept termination, abort, and swapping requests if the filtered service implements them.

Types of Filters

There are three types of filters:

- Replacement Filters

  A replacement filter performs the work required for a given request, then responds to the client. The request never reaches the filtered system service.

- One-way Filters

  When a one-way filter receives the request it filters, it performs some action based on the content of the request block, then forwards the request to the filtered system service, using ForwardRequest. The filtered service then responds to the client directly.

- Two-way Filters

  When a two-way filter receives the request it filters, it makes changes to the request block, then changes the exchResp field to its own exchange number. Then, it passes the request to the filtered system service, using RequestDirect.

  Because the filter's exchange number appears in the request block, the filtered service sends its response to the filter. The filter can then restore the client's exchange number to the request block, and respond to the client.

A filter can treat each request individually. Most filters will act as a one-way filter for some requests, and as a two-way or replacement filter for others.

Requirements for Filters

A filter should be invisible both to the client application and to the filtered service. To ensure this transparency, filters should observe the following guidelines:

1. One-way filters must not modify the client's request block. They will not have a chance to restore its contents later.

2. One-way filters must use ForwardRequest to pass the client's request block to the filtered service.

3. Two-way filters can modify the client's request block, but they must restore it to it's original form before they respond to the client.

4. Two-way filters must use RequestDirect to pass the client's request block to the filtered service.

5. Any filter that filters connection-oriented requests must serve the filtered service's operating system requests, and its deinstallation request.

Note on Keyboard Filters

All keyboard filters should filter the keyboard swapping request. The following example describes the consequences of a keyboard filter not performing a ServeRq on keyboard swapping requests.

The Context Manager maintains an outstanding ReadActionKbd request to the keyboard manager for **Action** key combinations. The key combination, **Action-Next**, for example, alerts the Context Manager to switch to a different context.

Assume that a two-way filter has been installed to intercept ReadKbd requests.

Under the Context Manager, a user is running an Executive program as the current context. The Executive is issuing a series of ReadKbd requests while the user types characters on the command line. The user types the characters **C**, **O**, and **P**, followed by the key combination **Action-Next**.

The Context Manager, whose priority is higher than the Executive, receives the **Action-Next** key combination before the filter receives the **P** keystroke. In response, the Context Manager initiates a swap to bring in the chosen context.

A swapping request is issued by the operating system. The request bypasses the filter and goes directly to the keyboard process, which responds.

The filter, which was not notified of the context switch, holds onto the ReadKbd request, waiting for the keystroke. The keyboard process will not send the keystroke, though, because it has received and responded to a swapping request for the client application. As a result, the swap fails with status code 813 ("Cannot swap out this partition").

System Services that Act as Agents

Agents are a special category of system service, and are used most often in communications software. Like filters, agents intercept requests before they arrive at their destination service. However, while a filter modifies a request and forwards it to a service on the same processor, an agent acts as a means to transport requests between processors.

Two types of agent exist; the client agent and the server agent. The client agent resides on the same processor as the client application. The server agent resides on the same processor as the system service the client needs.

Figure 8-1 shows the model of interaction between a client and a server through agents.

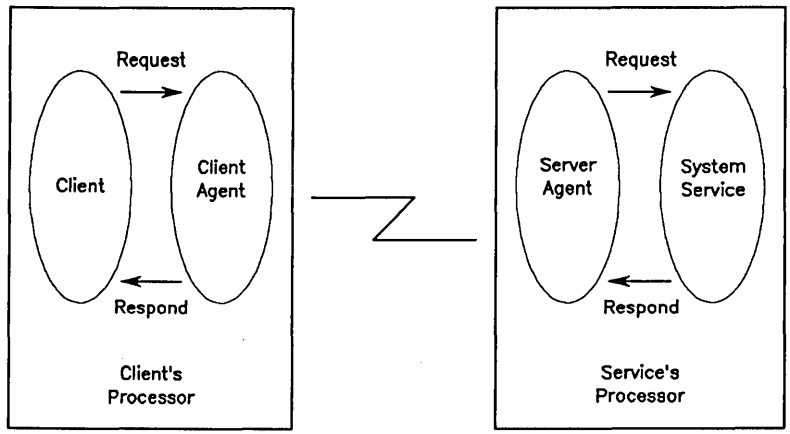


Figure 8-1. The Role of Agents in Request Passing

Role of the Client Agent

When the client agent receives a connection-opening or a connectionless request, it must determine which processor hosts the requested system service. It does this by parsing the data field pointed to by one of the request pbcb pairs in the request block. For example, it might parse the pbcb that identifies the file specification on a open file request. The request definition tables in the operating system identify which pbcb should be used.

The response to a connection-opening request always contains a handle from the remote service. The agent must save this handle for its own use, and allocate a new, local handle, for use by the client application. It then returns this new handle to the client.

When the client agent receives a subsequent connection-oriented request from the client, such as a read on a previously opened file, that request contains the local handle. The agent must then look up the handle it received from the remote service, and forward the request block to the service agent at the remote processor.

Finally, when it receives a close request, it must forward that request to the remote service. Then, after it receives a response from the remote service, it deallocates the remote connection handle, and responds to the client. This frees both the local and the remote connection handle for use on a subsequent connection.

In a network environment, the client agent must also track which remote processors a client has communicated with, and forward system requests for the client to all those processors. This allows the remote processors to deallocate resources when the client terminates. In addition, when it receives a response from a remote processor to a termination request, the agent must deallocate the handles for any resources on the remote processor that are owned by the terminating client.

Role of the Server Agent

The server agent acts on behalf of a remote client, by submitting the client's requests to the local operating system. To perform this role, the server agent must allocate local memory to store a copy of the client's request block, and any associated data.

When the remote client issues a request, the service agent receives the request block and request data from the remote client agent. It can then forward the request block to the local destination system service. When the server agent receives a response from the local system service, it sends the request block and the response data back to the client agent.

The service agent must also ensure that all user numbers are unique on its local processor. If a remote client has the same user number as a local partition, the service agent must map the client's user number to an unused local user number.

Request-Passing Guidelines for Agents

The transfer of requests between processors is optimized to minimize the amount of data that must actually be transferred. When a request is sent from a client agent to a server agent, the lower level network interfaces should send only the request block and the data pointed to by the request pbcb pairs.

When a response is sent from a server agent to a client agent, the lower level network interfaces needs to send only the request block and the data pointed to by the response pbcb pairs.

Implicit in these definitions is the fact that the contents of a request block may change as it passes through a network, and still remain valid. Therefore, agents should not count on receiving the same values in a request block that the client originally put there. Agents can, however, count on receiving consistent information in a series of requests. For example, all requests an agent receives for a given connection will contain the same connection handle, but the connection handle seen by an agent may be different from the one seen by the client application.

Piecemealing of Very Large Request Blocks

As described above, the service agent must provide local storage for the request blocks, request data, and response data. Therefore, it is possible for a request to be too large to fit in the service agent's buffers. This occurs most often for disk read and write requests, where using large blocks of data tends to optimize disk access time.

When a request is too large for its server agent to handle, the client agent should use *piecemealing*. Piecemealing consists of breaking a large request into several smaller requests, each of which is sent to the server agent individually.

In order to support piecemealing, a request must follow the format shown in Table 8-5. The request always has two pbcbs. The first one may be either a request or a response, depending on whether the operation performs a read or a write. The second is always a response pbcb. For piecemealing to occur, the request must also be defined as a piecemealable request in its :NetRouting: field in the request definition file.

Table 8-5. A Piecemealable Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0 or 1
3	nRespPbCb	1	1 or 2
4	userNum	2	User Number
6	exchResp	2	Response Exchange
8	ercRet	2	Status Code Returned
10	rqCode	2	Request Code
12	fh	2	File Handle
14	lfa	4	Logical File Address
18	pbData	4	Data Buffer
22	cbData	2	
24	pbCountRet	4	Returned Byte Count
28	cbCountRet	2	2

When the client agent detects a request that is too big for the service agent, and if the request can be piecemealed, it generates a series of requests small enough to fit in the service agent's buffer.

In each request, the client sends a piece of the original data, and changes the logical file address to indicate the position of that piece in the original data. As each response returns, the client agent adds the count of bytes processed to a running total.

Thus, after the server agent has responded to all the piecemealed requests, the client agent has a count of the total number of bytes processed.

If an error occurs in the processing of one piece of a piecemealed request block, the client application receives the error code. If more than one piece of the original request encounters an error, the client application

receives only the numerically largest error code. Because the client only issued one request, it can only receive one error code.

Defining Request Codes for a System Service

Almost all system services, except some filters, need one or more unique request codes, so that clients can make use of their services. The following sections describe how to define requests for a system service.

What You Need to Define

Before you can install your system service and let client applications begin using it, you must do two things. First, you must define your requests to the operating system, or it will not allow your service to serve them. Second, if you use a procedural interface you must create an object file that defines the names of your requests for the client application. Otherwise, the Linker will be unable to tie the symbolic name of your request to the operating system's request number.

To successfully define your requests to the operating system, you must perform the following three steps:

1. Choose your request codes. Requests that need a procedural interface must be defined in request level A, C, or E.

2. Define the format of your requests in a request definition text file (for example, ServerRequest.txt).

3. Create a binary request file (for example, ServerRequest.sys) from your request definition text file, using the Make Request Set utility.

4. Merge your binary request file with the operating system's request file, using the Install New Request utility.

For a client application to use your requests, you must also do the following:

1. Create a request label file that defines the procedure name for each of your request codes that has a procedural interface (for example, ServerRqLabel.asm).

2. Assemble your request label file into an object file (for example, ServerRqLabel.obj).

3. Link or Bind the object file into the client application.

Both these procedures are described in the following sections.

Defining a Request

The request definition text file defines all aspects of your system service's requests. The following sections explain the items you need to define. Listing 8-8 shows part of a sample request definition file. Each of the fields on the request definitions is defined below.

```
:WsAbortRq:0D905h
:TerminationRq:0D905h

:RequestCode:0EF00h
:RequestName:GetFooText
:Version:  2
: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:NoRouting

:RequestCode:0EF01h
:RequestName:DeinstallFooServer
:Version:  1
:LclSvcCode:0000h
:ServiceExch:exchInstalledMastr
:sCntInfo: 0
:nReqPbCb: 0
:nRespPbcb:0
:Params:   none
:NetRouting:noRouting
:SrpRouting:NoRouting

:RequestCode:0EF02h
:RequestName:OpenFooServer
:Version:  1
:LclSvcCode:0000h
```

continued...

```
:ServiceExch:exchInstalledMastr
:sCntInfo: 0
:nReqPbCb: 0
:nRespPbcb:1
:Params:   p(12), c(2,16)
:NetRouting:OpenFh, CloseAtTermination
:SrpRouting:NoRouting
```

Listing 8–8. Sample Request Definition File

The Structure of a Request

Each request in the request definition file is defined by a set of fields. Each field is identified by a field name, which is enclosed in colons. The contents of the field follows the field name, on the same line. The field names themselves are keywords recognized by the Make Request Set utility. Each request definition contains the fields described in Table 8-6.

Table 8-6. Request Definition Fields

(Page 1 of 2)

Field	Description
:RequestCode:	This field marks the start of the definition of a request. The value in this field is the request code number.
:RequestName:	Declares the name of the request, for documentation purposes. If the request has a procedural interface, the procedural name should be entered in this field. The procedural name is not read by the operating system.
:Version:	Defines the version number of the request (default = 0). This field controls the merging of request files. If two requests have the same request code, the one with the lower version number will be overwritten.

Table 8-6. Request Definition Fields

(Page 2 of 2)

Field	Description
:LclSvcCode:	Is used by the operating system for a special case but is generally not used in writing system services (default = 0). A value of 0DA31h indicates a dummy request. Note: You can put a signature value in this field when you define your request. Your service can then check for the signature at run time, using QueryRequestInfo. This allows you to verify that no other service has usurped your request code.
:ServiceExch:	Defines the exchange to which the request is routed before the first ServeRq operation for that request. This field can have either of two values: **ExchInstalledMastr** indicates that the request should be routed to the master if the service is not installed locally. **ExchInstalledLocal** indicates that the request should not be routed, and status code 33 should be returned if the service is not installed locally.
:sCntInfo:	Indicates the number of bytes of control information (default = 6).
:nReqPbCb:	Indicates the number of request pb/cb pairs in the request block.
:nRespPbCb:	Indicates the number of response pb/cb pairs in the request block.
:Params:	Defines the request procedural interface for the request. This field is used by the operating system for validation of request blocks. A fuller discussion of this field can be found later in this chapter.
:NetRouting:	Describes how the request should be routed in a networked environment. This field is described in more detail later in this chapter.
:SrpRouting:	Describes how requests are routed between processor boards on the Shared Resource Processor (SRP). This field is described in more detail later in this chapter.

System requests must be identified to the operating system, as well as defined. System requests should be identified at the beginning of the request definition file, then defined in the body of the file with the other requests. To identify a system request, you must enter the desired request code in the data field of the appropriate request definition directive. For example, :WsAbortRq:0EF05h.

Table 8-7 describes the system request directives.

Table 8-7. System Request Definition Directives

Directive	Description
:WsAbortRq:	Identifies the request code for an abort request.
:TerminationRq:	Identifies the request code for a termination request.
:SwappingRq:	Identifies the request code for a swapping request.

Defining the Procedural Interface

The :Params: field of each request definition defines the parameters for the procedural interface to the request. It contains one or more directives, each of which tells the operating system the size of one parameter, and that parameter's location in the request block. The parameters should be defined in the same order as they appear in the procedural interface.

When a client calls a system service via a procedural interface, it first pushes any required parameters onto the stack, then calls the procedure name. This passes control to the operating system, which builds a request block using the parameters pushed by the client. The operating system uses the information in the :Params: field to determine the size of each parameter, and its location in the request block. It then pops each parameter off of the stack, and copies it to the appropriate place in the request block.

The operating system has no way to check for correct use or number of parameters. The programmer who defines the procedural interface must make sure that valid parameters of the correct size are pushed in the correct order, or the request block the operating system creates will contain garbage.

The directives available for the :Params: field of the request definition are described in Table 8-8. Each :Params: field usually contains multiple directives, separated by commas. Note that there are five basic directives and five compound directives. The compound directives are most useful when sCntInfo is six.

Defining Request Routing

You must define two types of routing for each request: Network Routing, and SRP Routing. The :NetRouting: field defines how a request should be routed in a networked environment. The :SrpRouting: field defines how a request should be between processors on an SRP. Neither field affects routing across the local cluster.

The :NetRouting: field can contain multiple directives, separated by commas. The network routing directives are described in Table 8-9.

The :SrpRouting: field can contain only one directive. The SRP routing directives are described in Table 8-10.

Table 8-8. Request Procedural Interface Parameter Directives

Directive	Description
none	Indicates that there are no parameters for this procedural interface.
w(x)	The word at the top of the stack should be moved to offset x in the request block.
b(x)	The byte at the top of the stack should be moved to offset x in the request block. Note that while only a single byte is placed in the request block, a full word is popped from the stack.
p(x)	The double-word (pointer) at the top of the stack should be moved to offset x in the request block.
c(v, x)	The constant value v should be placed at offset x of the request block. No parameter is popped from the stack.
lfa	Equivalent to w(16), w(14). This directive specifies that the logical file address, which is stored in the second and third words of control information, is currently at the top of the stack.
pbcb0	Equivalent to p(18), w(22). This directive specifies that two parameters, a pointer and a word, should be popped from the stack and moved to locations 18 and 22 in the request block.
pbcb1	Equivalent to p(24), w(28). This directive specifies that two parameters, a pointer and a word, should be popped from the stack and moved to locations 24 and 28 in the request block.
pbcb2	Equivalent to p(30), w(34). This directive specifies that two parameters, a pointer and a word, should be popped from the stack and moved to locations 30 and 34 in the request block.
pbcb3	Equivalent to p(36), w(40). This directive specifies that two parameters, a pointer and a word, should be popped from the stack and moved to locations 36 and 40 in the request block.

Table 8-9. Request Network Routing Directives

(Page 1 of 2)

Directive	Description
NoRouting	Indicates that this request should not be routed outside the current cluster. When this directive is used, it must be used alone.
FileSpec	This request should be routed according to the file specification pointed to by the first pbcb pair in the request block. The default path should be inserted if the file name does not start with either a left brace ("{") or a left bracket ("[").
FileSpec2	Routing is identical to FileSpec. This directive tells the operating system that there is a second file name in the request block, which may need default path insertion.
FileSpecP2S2	This request should be routed according to the file specification pointed to by the second pbcb pair in the request block. The default path should be inserted if needed.
DevSpec	This request should be routed according to the device specification pointed to by the first pbcb pair. Routing is similar to FileSpec, but no default path is inserted.
DirSpec	This request should be routed according to the directory specification pointed to by the first pbcb pair in the request block. The default path should be inserted if the directory name does not start with either a left brace ("{") or a left bracket ("[").
rFh	This is a connection-oriented request, and must be routed according to the file handle contained in the first word of control information. This code can be used with FileSpec or DirSpec routing, to indicate that the actual routing should be by file handle, but that the operating system may need to insert the default path in one or more file specifications.
SpecPW	Indicates that the default password may need to be inserted. When used with file, directory, or device routing, each name pbcb is assumed to have a pbcb immediately following it which points to a password. Otherwise, the first pbcb in the request block is assumed to point to a password.

Table 8-9. Request Network Routing Directives

(Page 2 of 2)

Directive	Description
RW	This is a read or write request and may be piecemealed. This code should be used only if the request conforms to the required format for piecemealable requests.
OpenFh	Indicates a connection-opening request. The resource handle should be returned in the first response pbcb in the request block.
CloseAtTermination	This directive should be used with the OpenFh directive. It indicates that the new connection is non-permanent and should be closed if the client terminates. If this directive is not included in an OpenFh request, the connection remains open after the client terminates.
CloseFh	Indicates a connection-closing request, and tells the operating system that the corresponding file handle should be released. The rFh directive should also be included for connection-closing requests.

Table 8-10. Request SRP Routing Directives

(Page 1 of 2)

Directive	Description
NoRouting	SRP routing is unspecified. The operating system defaults to rRemote.
rLocal	Indicates that the request should be served on the same processor.
rRemote	The request is served locally if possible, else it is routed according to the SRP request routing table.

Table 8-10. Request SRP Routing Directives
(Page 2 of 2)

Directive	Description
rHandle	The request is routed by file handle. Routing is similar to network routing by handle, but the mechanism applies only to file system requests. NOTE: This routing scheme requires special handles. See your operating system documentation for details.
rFileId	The low-order byte of the handle identifies the processor to which the request should be routed. NOTE: This routing scheme requires special handles. See your operating system documentation for details.
rLineNumber	Route to the processor which owns the cluster line specified in the handle. NOTE: This routing scheme requires special handles. See your operating system documentation for details.
rDevice	Route the request by handle or by file specification to the processor specified by the routing name. The routing name serves as a key to look up the target processor in the Master FP Name Table.

Making a Loadable Request Set

After the requests have been defined in a request definition file, they must be merged with the operating system on which the service or its client will run. Software products that use requests should merge those requests as part of their installation script when they are first installed on a user's hard disk.

Before the requests can be merged, they must be converted to binary form, using the Make Request Set utility.

After the requests have been converted to binary form, they can be merged with the operating system, using the Install New Request utility.

For more information about these utilities, see your operating system documentation.

Making a Request Label Object File

Finally, to allow the client application to call the request procedural interface by its symbolic name, you must create an object file that defines the name, and link that file with the client application.

Most implementations of CTOS/Open provide an Assembler macro, *%RqName*, which aids in defining request labels. This macro is generally located in the file *RqLabl.mdf*. See your operating system documentation for more information. Listing 8-9 shows an example of a request label file for a system service.

```
; request labels for the requests in FooServerRq.c
; FooServerRqLabel.asm

$INCLUDE([Sys]<Sys>RqLabl.mdf)

%RqName( 0EF00h, "GetFooText")
%RqName( 0EF01h, "DeinstallFooServer")
%RqName( 0EF02h, "OpenFooServer")
%RqName( 0EF03h, "CloseFooServer")
```

Listing 8–9. Sample Request Label File

9

Writing System-Common Services

Introduction

Two types of system services exist: request-based system services, and system-common services. The two are somewhat similar, in that each performs a service for a client application, but the methods by which the two types perform the service are quite different.

Request-based system services are described in the previous chapter. This chapter describes system-common services.

Request-Based vs. System-Common Services

Request-based system services and system-common services are suited to different types of work. This section describes how the system-common model differs from the request-based model, and gives guidelines for deciding which model is more appropriate for a given task.

The System-Common Model

The system-common model is different from the request-based model in one major way.

In the request-based model, the client sends a request to the service. The service then takes over control of the processor and performs its work. While the system service executes, the client process waits. When the service has finished, it sends a response to the client. This allows the client to regain control of the processor and begin executing again.

In the system-common model, there are no requests or responses. When the system-common service installs itself, it tells the operating system the

entry point and parameters of each of its system-common procedures. After it has defined its system-common procedures to the operating system, the service simply waits to deinstall. It performs no more work, itself.

To use a system-common procedure, clients simply call the procedure as if it were part of the client program. The operating system detects that the called procedure is actually a system-common procedure, and transfers the client's execution point to the beginning of the system-common procedure.

The client never gives up control of the processor. The system-common procedure executes as part of the client process. Then, when the system-common procedure returns, the operating system resets the client's execution point to the appropriate place in the client program.

Figure 9-1 shows the threads of execution in a request-based system service model, and in the system-common service model.

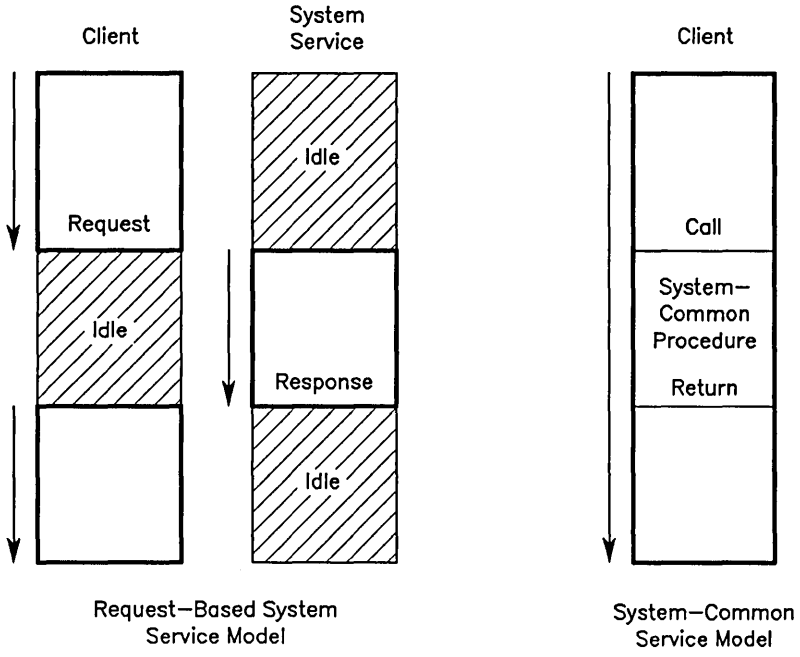


Figure 9–1. Threads of Execution in System Services

In summary, a system-common service consists of one or more globally accessible procedures, which are managed by the operating system. When a client calls a system-common procedure, that procedure executes as if it were part of the client application. The system-common service itself performs no work other than installing and deinstalling the procedures it controls.

Special Features of System-Common Procedures

System-common procedures have two features that give them an advantage over request-based system services under some circumstances:

- System-common services eliminate the need to define requests, except for the service's deinstallation request.
- Because system-common services do not use requests, they eliminate the overhead associated with the request mechanism.

Eliminating the need to define requests is mainly a point of convenience. This feature can also make system-common services easier to install on a hard disk, because they often eliminate the need to merge request files.

Because system-common procedures eliminate the task switches and other overhead required by the request mechanism, they have a significant performance advantage over request-based system services. This can save a substantial amount of execution time, especially for short procedures.

Requirements for System-Common Procedures

Because system-common procedures execute as if they were part of the client application, there are several requirements they must meet.

- A system-common service must reside on the same processor as its client.
- All system-common procedures must be reentrant.
- System-common services must use selectors from the processor's Global Descriptor Table (GDT).

The major restriction on system-common procedures is that a system-common service must reside on the same processor as its client. Because system-common procedures do not use the request mechanism, there is no way for a client to communicate with a service that resides on different processor.

All system-common procedures must also be reentrant. A reentrant procedure is one which may be entered repeatedly, and may be entered before prior executions of the same procedure have been completed. Neither its external parameters nor its instructions can be modified during its execution.

System-common procedures must be reentrant because clients execute the actual code in the system-common procedure, and a system-common service can have multiple clients. Therefore, one client may begin executing a procedure before the previous client has finished executing it.

All system-common procedures must use the processor's Global Descriptor Table (GDT). This allows the client to execute code that resides in the system-common procedure without requiring the operating system to perform any selector aliasing.

In order for a program to use the processor's GDT, the program must be linked using the Bind command, with a run file mode of "GDTProtected."

Deciding Which Type of Service is Appropriate for Your Task

Request-based services and system-common services are each suited to different circumstances. When clients may need to access the system service across the cluster or over a network, the service must be request-based. However, if the service will always reside on the same processor as its client, a system-common service can provide better performance.

Writing System-Common Procedures

A system-common service can be thought of as a collection of independent procedures. Each procedure has its own entry point, and has no necessary relationship to any other procedure. In practice, the procedures in a system-common service are usually related to each other, because they usually perform similar or related operations.

The procedures in a system-common service may also reference the service's global data, to obtain status or other information. If a system-common procedure needs access to the service's global data, the procedure needs to load the service's global data segment address in its data segment register. If the service is compiled using the Large model of computation, this occurs automatically. See your programming language documentation for more information.

System-common procedures should generally not write to the service's global data, because that would violate the rules for reentrance. System-common procedures should only write to local variables and to memory addresses passed as parameters by the client.

Listing 9-1 shows an example of a system-common procedure which performs a task similar to the one in the request-based system service from Chapter 8. The pragma, CTOS_CALLING_CONVENTIONS, informs the compiler that the procedure should use the standard CTOS calling convention, described in Part II of this manual, instead of the C language calling convention.

The variables *rgMsgRet* and *cbMsgRet* are a character string and its size, located in the service's global data.

```
/* Process calls to our system-common procedure */
pragma Calling_convention(CTOS_CALLING_CONVENTIONS);
ErcType GetFooText(Pointer pbDataRet, Word cbDataRet,
                   Pointer psDataRet)
[
Word    i=0;

/* get the maximum size of the message */
if(cbDataRet <  cbMsgRet)
    i = cbDataRet;
else
    i = cbMsgRet;

/* put the message in the user's buffer */
memcpy(pbDataRet, rgMsgRet, i);
*((Word *) psDataRet) = i;

/* return ercOK */
return(0);
}
pragma Calling_convention();
```

Listing 9–1. Sample System-Common Procedure

Defining Parameters for a Procedure

When a system-common service installs one of its system-common procedures, it passes two things to the operating system: the system-common procedure number, and a character string that defines the parameters for that procedure.

This section explains the format of the parameter-definition string. System-common procedure numbers are discussed later in this chapter, under "Defining System-Common Procedure Numbers."

System-common services should supply a parameter-definition string for each system-common procedure they install. If a procedure takes no parameters, the parameter-definition string should be a null pointer with a zero length.

The syntax for defining system-common procedure parameter lists is shown in the following paragraphs. This syntax is in Backus–Naur Form (BNF) language.

```
<definition string> ::= <definition string><term>

<term> ::= <word term>
         |  <selector term>
         |  <pointer term>
         |  <pointer term><count term>
<word term> ::= w
<selector term> ::= s
<pointer term> ::= <input pointer term>
                |  <returned pointer term>
<input pointer term> ::= p
<returned pointer term> ::= q
<count term> ::= c
```

For example, for the call

```
GetFooText(Pointer pbDataRet, Word cbDataRet, Pointer
            psDataRet)
```

the string defining the parameter list would be "pcp" where the calling sequence is "pointer, count, pointer." The operating system can then use this string to create alias pointers, if it needs to. For example, when the operating system intercepts (traps) a call made by a real mode program to the installed procedure, the operating system creates an alias GDT selector for each pointer in the parameter-definition string.

The operating system may modify this string for its own purposes internally, but it will not change the meaning or the order of the parameters. Any changes are purely for the operating system's internal use. A system-common service should not, however, expect to receive the same parameter definition string on a SystemCommonQuery that it passed on a previous SystemCommonInstall.

Installation and Deinstallation

System-common services can be installed and deinstalled, much like request-based system services. Usually, a system-common service serves only a single request. That request is issued by the service's deinstallation utility, and is used to deinstall the service.

Installation

A system-common service uses almost the same installation procedure as a request-based system service does. The key difference is that instead of serving a request for each of its major functions, a system-common service installs each of its system-common procedures.

The system-common service first calls SystemCommonQuery for each procedure to determine whether the desired system-common procedure number has already been declared to the operating system. This step is even more important that the QueryRequestInfo call in a request-based service. This is because a system-common service performs no process comparable to merging request files for a request-based service. Therefore, if two system-common services use the same system-common procedure number, that fact may not be detected until run time.

As with a request-based system service, a system-common service should save the existing state of its system-common procedure numbers, then restore them when it deinstalls.

A typical installation sequence is:

1. Call GetPartitionHandle to see if the service is already installed in another partition. A system service should not install if it is already present on the current processor.

2. Allocate any resources to be used by the system-common service. This includes allocating exchanges, allocating memory, and creating any additional processes.

3. Query the current status of the deinstallation request the service will serve. This lets the service make sure the request is declared in the operating system. If the request is undeclared, the service should abort. The service should also save the values returned by the query so that it can restore them when it is deinstalled.

4. Query the current status of the system-common procedure numbers the service will use. This lets the service make sure they are not already in use. The service should save the values returned by the queries so that it can restore them when it is deinstalled.

5. Perform ConvertToSys, then Exit. The ConvertToSys operation changes all processes, exchanges and memory in the partition from

application status to system service status. This converts the partition to a system service partition. The Exit operation then reloads the exit run file into memory.

6. Change priority, so that the service runs at a higher priority than any potential client application.

7. Use the SetPartitionName operation to identify the service to the operating system.

8. Perform a ServeRq operation for the deinstallation request.

9. Perform a SystemCommonInstall operation for each of the service's system-common procedures.

As with a request-based system service, the order of the operations before ConvertToSys can be changed, and the order of the operations after ConvertToSys can be changed. Compatible programs generally should not, however, move an operation from before to after ConvertToSys, or from after to before.

Listing 9-2 shows an example of how a system-common service might install itself. The service installs its system-common procedures, then waits at its exchange for a deinstallation request.

```
void main(){
RqHeaderType   *pRq;

    CheckErc(GetUserNumber(&usernum));
    CheckErc(AllocExch(&exchServe));

/* This is the already-installed test.  If already */
/* present, set up an exit message, and exit */
    if (GetPartitionHandle(rgServerName, srgServerName,
                        &PhRet) == ercOK) {
        ErrorExitString(0, rgInstMsg1, srgInstMsg1);
    }

/* query the system-common procedure.  If it is in */
/* use, error exit. */
    CheckErc( SystemCommonQuery(IDSYSCOM, &rgSysComInfo,
              sSysComInfo));
    if(rgSysComInfo.fDefined != FALSE)
        ErrorExitString(0, rgRqInUse, srgRqInUse);
```

continued...

```
/* query the request.  If it is in use, error exit. */
    CheckErc( QueryRequestInfo(DEINSTRQ, &rqInfo,
                sizeof(rqInfo)) );

/* BECOME A SERVER */
/*  Since the server is not already installed, install */
/*  now.  Set up an appropriate exit message, and */
/*  convert to sys (become a server). */
/*  The ErrorExitString call after the convertosys call*/
/*  allows the reloading of an exec in the primary */
/*  partition. */
/*  Note: you must do setmsgret before convertosys.  */
/*  Can't print a string to the screen after */
/*  converttosys. */
    CheckErc(SetMsgRet(rgInstMsg2, srgInstMsg2) );
    erc = ConvertToSys();
    if(erc != 0) {
        ErrorExitString(erc, rgInstErrorMsg,
                        srgInstErrorMsg);
        }
    else
        Exit(0);

/*  All servers must run at a priority higher than 40, */
/*  else deadlocks can occur. */
    preTerminateIfErcNotOk(ChangePriority(12));

/*  Set the partition name so that the deinstallation */
/*  program can obtain this partition handle. */
    preTerminateIfErcNotOk(SetPartitionName(0,
                    rgServerName, srgServerName));

/*  Now serve the Request to the service exchange for */
/*  this program. */
    preTerminateIfErcNotOk(ServeRq(DEINSTRQ, exchServe));
    preTerminateIfErcNotOk(SystemCommonInstall(IDSYSCOM,
                            (Pointer) GetFooText, rgParams,
                            srgParams, 0, NULL, 0));

/*  Server loop... */
/*   Check for a request.  If there is one, deinstall. */
    while(TRUE) {
        Wait(exchServe, &pRq);
        DeInstall(pRq);
        }
}
```

Listing 9–2. System-Common Service Installation

Deinstallation

The procedure to deinstall a system-common service is similar to the one for a request-based service.

A system-common service generally serves only a deinstallation request, so it need not concern itself with outstanding requests. However, there is a deinstallation issue raised by the nature of system-common procedures: a client may be currently executing the procedure when it is deinstalled.

If this happens, the client's selector is suddenly invalid. The next instruction the client program executes will cause a protection fault, and the client program will fail. For this reason, a system-common service should only be deinstalled if none of its clients are active.

When a system-common service receives a deinstallation request, it should:

1. Restore the request table in the operating system to its original state for the service's deinstallation request, by issuing a ServeRq operation with the saved value from installation time.

2. Restore the system-common procedure table by calling SystemCommonInstall with each saved value from installation time.

3. Close any connections the service may have opened as a client (with the file system, for example).

4. Unlock its own partition by calling SetPartitionLock, then respond to the deinstallation request.

5. Terminate by calling ExitAndRemove.

Listing 9-3 shows the deinstallation procedure for a system-common service.

```
void DeInstall(RqHeaderType *pRq) {

/* unserve the requests */
CrashIfErcNotOk(ServeRq(DEINSTRQ, rqInfo.exch));
CrashIfErcNotOk(SystemCommonInstall(IDSYSCOM,
       rgSysComInfo.pProc, rgSysComInfo.rgbParamDef,
       rgSysComInfo.srgbParamDef, 1, NULL, 0) );

/* All done, unlock partition, and deinstall  */
    CrashIfErcNotOk (SetPartitionLock(FALSE));
                          /* unlock partition */
    Respond(pRq);         /* respond to deinst. rq */
    ExitAndRemove();      /* bye  */
    Crash(3);      /* Bad problem if we ever get here */
}
```

Listing 9–3. System-Common Service Deinstallation Procedure

Defining System-Common Procedure Numbers

Like requests, system-common procedures each have a code number that identifies them to the operating system. For applications to use them, each system-common procedure also needs to have a symbolic name.

What You Need to Define

For a client program to call a system-common procedure, it must know the symbolic name for that procedure. To define symbolic names for your system-common procedures, you must create a system-common label object file, then link that file with the client application. This way, the client application can simply call the system-common procedure by name, as an external procedure. For example:

```
GetFooText(&rgbServerData, cbServerData, &sDataRet);
```

System-common procedures can use procedure numbers 6000h through 7FFFh without restriction. You can also reserve system-common procedure numbers in the range from 4000h to 5FFFh by contacting Unisys Network Computing Group, CTOS/BTOS Technical Support.

Making a System-Common Label Object File

Most implementations of CTOS/Open provide an Assembler macro, *%OsSubLab*, which aids in defining system-common procedure labels. This macro is generally located in the file *OsSub.mdf*. See your operating system documentation for more information. Listing 9-4 shows an example of a system-common procedure definition file for a system-common service.

```
STACK SEGMENT STACK 'STACK' ; default stack
    DB 512 DUP (?)
STACK ENDS

$include([Sys]<Sys>ossub.mdf)

%OsSubLab(26005, "GetFooText")
```

Listing 9–4. Sample System-Common Label File

Part II – Advanced Topics

10

Stack Format and Calling Conventions

This chapter describes the segment ordering, stack format, and calling conventions used with CTOS.

Introduction

Several compiled languages and assembly language are supported on the CTOS-based operating systems. These operating systems and their system libraries use a medium model of computation and a conventional stack format for all procedure calls.

As a historical note, these conventions were derived from the PL/M-86 programming language, in which the original implementation of CTOS was written.

Memory Addressing

Memory addressing is described in detail in Chapter 5, "Protected Mode Programming Guidelines" in Part I of this manual. As a brief review of that section, recall that CTOS addresses memory using a 32-bit segmented address. The high-order 16 bits contain the part of the address called the *segment selector*, which identifies a particular segment in memory. The low-order 16 bits contain an *offset* within that segment. A memory address is written as follows:

```
segment selector:offset
```

The meaning of the segment selector value differs, depending upon whether the operating system is a real mode operating system or a protected mode operating system. In real mode, a segment selector refers to a specific paragraph in a 1-megabyte address space. In protected mode,

a segment selector is simply an index into a descriptor table maintained by the processor. There is no relation between a protected-mode selector and any location in physical memory.

Individual bytes within a segment are addressed by the offset from the segment selector. Because the offset contains 16 bits, there are 64K possible addresses within the segment.

Program Segmentation

Generally, a program is written in several separate source files. Each of these source files is compiled into an object module. An object module is composed of one or more *segment elements* of different *classes*. The classes used in an object module are typically code, static data, constant data, and stack.

When the object modules are linked, the Linker follows a set of ordering rules to generate a run file. The Linker sorts and recombines the segments in each object module by class to produce larger *linker segments*. From this point on, these linker segments are referred to simply as *segments*. (For a more detailed discussion of the Linker's function, see your operating system documentation.)

When the program executes, each of these segments receives a unique selector, which identifies the area of memory in which that segment resides. When the program calls a procedure or accesses data, the processor updates its registers, so that the called procedure or data can be addressed.

If the new address is in the same segment at the previously-used one, the processor only needs to update the register that contains the offset of the requested item. When only the offset must be changed, the event is called a *near reference* (for data) or a *near call* (for a procedure).

If the new address is in a different segment from the previously-used one, the processor must load both the selector and the offset of the desired item. This requires the processor to change the contents of two registers, instead of one. This requires more time than changing just the offset. When both the segment and the offset must be changed, the event is called a *far reference* (for data) or a *far call* (for a procedure).

Most programs that run under CTOS use the *medium model* of segmentation. In the medium model, all calls are *far* and all references to static data and the stack are *near*.

Memory Organization

The Medium Model

As noted in the previous section, the medium model of segmentation is the most common one used by CTOS programs. One of the requirements of the medium model is that the total size of the program's static data segment, constant data segment, and stack segment must be less than 64K bytes. In other words, all data in those three parts of the program be addressable as offsets from one segment register (normally DS).

This requirement exists because, in the medium model, these three segments are defined as a *group*. A group is simply a collection of segments that total less then 64K bytes in size. Grouping the segments allows all the data in them to be addressed as offsets from a single segment register.

In the medium model, the static data segment, the constant data segment, and the stack segment are defined as a group called *DGroup*. Hence the size restriction. Unless you program in assembly language, you don't need to define DGroup yourself – your compiler does it for you.

Why group segments? Why not address them each individually? The answer is speed. Loading a segment register takes time, so the less often a program has to load one, the faster it runs. This effect is especially great when a segment is referenced frequently during program execution, as the ones in DGroup are.

Figure 10-1 shows typical memory organization for a medium model program. The order of the data, constant, and stack segments within DGroup can vary. For details, see "Changing Memory Organization From the Default for Your Compiler," later in this section.

In real mode, code segments and the data/stack segments (DGroup) are loaded adjacent to each other in memory, with DGroup normally positioned at higher addresses than code. This organization matches the one shown in Figure 10-1.

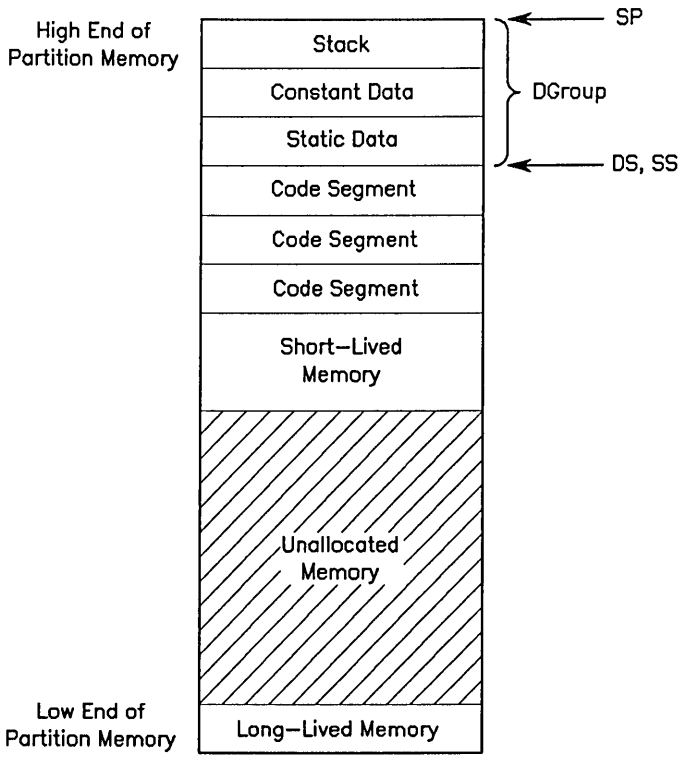


Figure 10–1. Typical Memory Organization in Medium Model Programs

In protected mode, however, code is loaded independently of data and may be located anywhere in memory. In protected mode, the program does not need to know where its code is physically located. The operating system manages all the addressing required.

The medium-model memory organization originated with the PL/M compiler, and most languages conform to it. See your programming language manual for more information.

Code Segments

Any number of separate code segments, each of which may contain up to 64K bytes, are addressed based on the CS (Code Segment) register. Only one code segment, the currently executing one, is addressable at a time. The current code segment (and therefore the value in the CS register) changes with each far procedure call, far jump, or far return. All calls, jumps, and returns are far in the medium model, unless explicitly overridden. The IP (Instruction Pointer) register contains the offset of the next instruction to be executed.

Figure 10-1 shows DGroup positioned just above and adjacent to the program's code segments. This representation is typical in real mode. As mentioned earlier, in protected mode, code segments are loaded independently and may be located anywhere in memory.

Unallocated Memory

In real mode, a pool of unallocated memory is located below and immediately adjacent to the program, as shown in Figure 10-1. In protected mode, however, the memory pool is usually located just below and adjacent to DGroup, while the program's code may be anywhere, as was described earlier in this section. Short-lived memory is dynamically allocated toward lower addresses from the higher end of the memory pool. Long-lived memory is allocated toward higher addresses from the lower end of the pool. (For details, see the discussion of memory management in your operating system manual.)

Notes on the Stack

Initially, the value in the SP (Stack Pointer) register is the highest address in the stack segment: the top of the stack. The value in SP decreases as the stack grows downward in memory.

Most compilers automatically provide more than enough stack space for their programs, but some programs may need to allocate extra space. This is more likely to be needed when a program performs many nested procedure calls (for example, if it uses recursion).

Caution: *If the stack requirements of the program exceed what is allotted, the growing stack can overwrite program data, which may cause unpredictable results. If your program links successfully but malfunctions, and you suspect stack overflow, increase the stack size.*

Values of DS and SS in Medium Model

In medium model, the DS (Data Segment) and SS (Stack Segment) registers have the same value. Both contain the selector that identifies DGroup.

In a program, the "objects" that you manipulate are usually static constants, static data variables, and the local variables on the stack. (See "Procedure Calls and the Stack," later in this section, for details on the stack format.) It is most efficient to address all three using the same segment selector. As a simple example, assume that A is a static data variable, B is a local variable, and C is a constant in the following expression:

A = B + C

If all three objects can be addressed by the same segment selector, only the offset of each object is needed to do the calculations. If they cannot all be addressed by the same segment register, then the processor incurs more addressing overhead.

If DS and SS are equal, and an object on the stack (referenced from SS) is assigned to a static variable (referenced from DS), there is no segment register overhead required.

Caution: *A common error in writing a multiprocessing program is to create a stack that fails to follow the convention of equating DS and SS. This error has no effect as long as you are fetching or storing the values of variables. However, it can cause invalid pointers to variables. Invalid pointers cause protection faults.*

Changing Memory Organization from the Default for Your Compiler

High-level language compilers produce program modules in which the segment elements appear in a characteristic order by segment class (such as Code or Data). The Linker accepts the class order of the first object module it encounters as the order for the entire program.

Under some circumstances (for example, when using DS allocation, which is discussed next), you may need to change the order in which program segments are organized in memory. To do so, you must write an assembly language module that does nothing except declare segments in the order you want them. Then, list this template module first in your input to the Linker. (Such a template module may already be available with your language, under the file name of First.obj.)

The following is an example of such a template module:

```
stack segment word 'stack'
stack ends
data segment 'data'
data ends
const segment 'const'
const ends
memory segment 'memory'
memory ends
dgroup group stack,DATA,const,memory

code segment 'code'
code ends

end
```

Code Sharing

Most compilers emit code that supports the code-sharing feature of CTOS. The only requirement for code sharing is that code be separate from data. If this requirement is met, then multiple instances of the same program can share a single set of code segments. Each new instance of the program receives a new data segment, and selectors for the existing code segments.

Disposable Initialization Code

CTOS allows two class names for executable code segments: CODE and COED. Most programs use only CODE segments, which are normal, executable code segments. Some programs, however, need code which can be used once and then thrown away.

For example, a system service has no need to keep its installation code after it has installed itself. To conserve memory, a system service writer could put the installation code in a COED segment, then deallocate that segment after the service installs itself.

Creating a COED Module

If you program in Assembly language, creating a COED segment is easy. Simply define a segment with class name COED and put your installation code in it. Then make that segment the first one in your program's memory image (in other words, at the lowest addresses in memory).

If you program in a high-level language, however, you need a utility to convert the class name in the object module your compiler emits. Most implementations of CTOS provide a utility that performs this function. See your operating system documentation for more information.

To create a COED segment in a high-level language:

1. Create a small Assembly language template file that defines the COED segment as the *first* segment in the run file. See "Changing Memory Organization from the Default for Your Compiler" for details.

2. Put the disposable code in a separate source file from the rest of your code.

3. Compile that file to an object module.

4. Run the conversion utility on that object module.

5. Link the disposable code module with the rest of your program. Be sure to specify your segment template file as the first file on the Linker's input line.

The run file you create should now have a COED segment at the bottom of its memory image, which contains the disposable code.

Disposing of the Code in a COED Module

COED segments are treated as if they were dynamically allocated memory. For this reason, the COED segment must be disposed of *before* the program allocates any dynamic memory.

After the program has executed the disposable code, it should release the COED segment that contains the code. To do so, the program can simply call DeallocMemorySL. For example, a COED segment that contains the procedure *Initialize* can be deallocated as follows:

```
pCOEDSeg = NULL;
selectorof(pCOEDSeg) = selectorof(Initialize);
CheckErc(DeallocMemorySL( pCOEDSeg, 0xFFFF) );
```

The above code fragment first creates a pointer to the segment which contains the Initialize procedure (the COED segment), with an offset of 0. Next, the fragment calls DeallocMemorySL with the pointer to the COED segment, and the hexadecimal value 0xFFFF. These parameters tell the operating system to deallocate the entire segment.

Using DS Allocation

DS allocation is a Linker option, which allows a program to "pre-allocate" some memory that can be addressed using DS. The following sections discuss the memory organization of a program that uses DS allocation. For information about how to access the memory reserved by DS allocation, see Chapter 16, "Memory Management," later in this manual.

The following discussion assumes that the medium model of segmentation is used.

What Is DS Allocation?

DS allocation allows your program to address parts of memory not occupied by the program with offsets relative to DS. Your program can

then use these parts of memory for dynamic data objects (such as the separate stacks for several processes) and still address them efficiently.

DS allocation causes the Linker to create an *expand-down* segment for DGroup. Expand down segments are of maximum segment size (64K bytes) and grow towards lower addresses. Thus, when DS allocation is used, the operating system's program loader positions the highest byte of data in DGroup at offset 0FFFFh, allowing DS to be the base for a full 64K-byte segment.

DS allocation presents different challenges in real mode and in protected mode.

DS Allocation in Real Mode

In real mode, code and data are loaded contiguously with code normally positioned below and adjacent to data, as was shown in Figure 10-1. If you were to specify DS allocation in conjunction with the memory organization shown in Figure 10-1, and if DGroup occupied less than a full 64K byte segment, the DS (and SS) would point to some location in a code segment (or in the memory pool just below the code segments).

Because of this, you may need to reorder your program segments so that DGroup resides at the lowest program addresses. To do this, you need to write an assembly language program, as described in "Changing Memory Organization From the Default for Your Compiler," earlier in this chapter.

A typical example of memory arrangement using DS allocation is shown in Figure 10-2. This arrangement places the value of DS and SS somewhere in the pool of unallocated memory. During execution, the program can allocate and deallocate short-lived memory within the range of DS and address it with offsets.

DS Allocation in Protected Mode

Because protected mode operating systems load code and data separately, you just need to notify the Linker that you want DS allocation. An assembly language module is not needed.

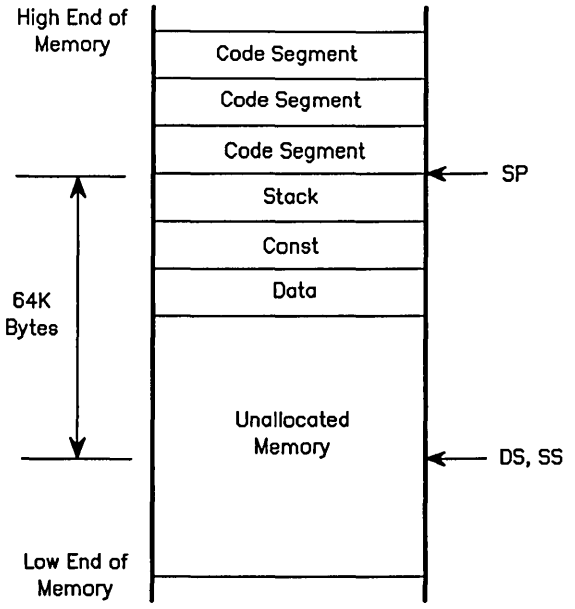


Figure 10-2. DS Allocation

Procedure Calls and the Stack

Overview

A procedure (subroutine) call requires interrupting the sequential execution of code, continuing execution at a different location, then returning to the point from which the call was made. When one procedure calls another, therefore, the environment must be saved so that it is possible to return later. Typically, a procedure call also requires passing some input parameters.

The program's stack is used to fulfill both of these requirements. The stack also may be used by interrupt and trap handlers to save the context of a process when a system interrupt occurs. (See the discussion of interrupt handlers in your operating system manual for more information on interrupts.)

Parameter Passing

Parameters may be passed either by reference or by value. When a parameter is passed by value, a copy of the parameter's current value is placed on the stack for access by the called procedure. Such a parameter can only be a 1-, 2-, or 4-byte quantity, because those are the sizes that the stack can accommodate. When parameters are passed by value, the called procedure cannot affect the original value of the parameter in the caller's data space. The called procedure can only change the copy it received.

When a parameter is passed by reference, the *address* of the parameter, not its current value, is placed on the stack. In this way, a called procedure can access a parameter of any size, despite the stack's size limit of 1, 2, or 4 bytes on the value actually passed. When parameters are passed by reference, the called procedure *can* change the contents of the parameter in the caller's data space.

Different languages use different parameter-passing conventions. The procedures in CTOS expect parameters to be passed by value. If a parameter is a word (wMyWord, for example), the procedure expects a word value, not a pointer to a word value.

Standard Stack Format

The standard medium-model stack format for CTOS is shown in Figure 10-3. This stack format originated with PL/M, and most available high-level language compilers adhere to it. Each location shown on the stack is a 2-byte word.

The initial value of the SP (stack pointer) register is at the highest address of the stack. The stack grows down toward lower addresses as objects are pushed onto it. Thus, as the stack grows, the address of the top of the stack (SP) becomes smaller.

When a procedure is called, the first thing it should do is execute a few instructions to set up its stack frame. These instructions are sometimes called a prolog.

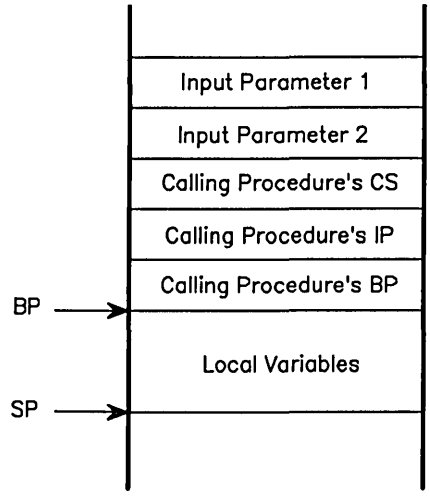


Figure 10–3. Standard Stack Format

The format shown in Figure 10-3 is how the stack appears *after* this standard prolog has been executed. Certain operating system components, such as the Virtual Code Management facility and the Debugger, trace procedure calls through the stack. If the stack format is nonstandard, these features will not work correctly.

The standard stack format allows parameters and local variables to be referenced in a simple, standard way for all procedures. The description of the standard prolog and epilog, below, explain why.

Standard Prolog and Epilog

All procedures should contain a standard prolog and epilog. The prolog sets up the procedure's stack frame. The epilog removes it. The standard prolog and epilog are shown at the top of the next page.

```
PUSH BP              ;Prolog
MOV BP,SP
SUB SP,n
.
.
.
MOV SP, BP           ;Epilog
POP BP
RET m
```

In the prolog above, the value of BP (the base of the caller's stack frame) is pushed onto the stack, after which the value of SP is copied to BP, establishing a new stack frame.

Next, the SUB SP,n instruction subtracts *n* bytes of stack space for local and temporary variables from the value of SP. This allows the called procedure to address input parameters and local variables relative to BP. Input parameters have a positive offset from BP (the first parameter is at BP+6). Local variables have a negative offset from BP (the first local variable is at BP–2).

In the epilog, the stack pointer (SP) is first set equal to the base pointer (BP), eliminating the local variables from the stack. This step also restores to SP the value it contained after the PUSH BP instruction in the prolog.

Next, the value of the caller's BP is popped from the stack. This step restores BP to the value it had on entry to the procedure.

Finally, the RET *m* (return to caller's CS:IP) instruction tells the processor to return to the caller, and to pop *m* bytes from the stack when it does so. This returns control to the caller, and sets SP to point to value it contained before the caller pushed any input parameters onto the stack.

Note that in code generated by most C compilers, where a variable number of arguments can be passed, the epilog differs slightly. The value of *m* on the RET statement is always 0 in the C language, because the called procedure cannot know in advance how many parameters it will receive. In the C language, the caller pops its own parameters when control returns to it. See your C compiler manual for details.

11

Mixed-Language Programming

This chapter describes the issues that need to be addressed when a program contains object modules built by different language compilers.

Issues in Mixed-Language Programming

Object modules whose source code was written in different programming languages can be linked together and can work successfully. However, doing so requires somewhat more work than linking object modules whose source code was written in the same language. This extra work consists of resolving any differences in the way the two programming languages handle procedure calls and returns.

Two common cases exist where a program may include object modules compiled from different source languages:

- When a program calls a CTOS library procedure from a language other than PL/M.
- When a program calls a subroutine which was written in another language. For example, a FORTRAN program might call a customized library function which was written in Pascal.

The following sections describe the issues that may need to be addressed before such calls can work properly.

Parameter Passing and the Stack

In general, a called procedure expects its input parameters to be of specific sizes, and located in specific positions on the stack. Therefore, if a called procedure expects a two-byte parameter, the calling program must be sure to pass a two-byte parameter to that procedure.

The calling program must also be sure to pass the correct number of parameters, in the correct order. Some languages push parameters in a left-to-right order, so that the leftmost parameter is the "bottom" parameter on the stack. Other languages push parameters in a right-to-left order, so that the leftmost parameter is the "top" parameter on the stack.

In addition, some languages pass parameters by value, others pass parameters by reference. When a language passes a parameter by value, it passes the *current value* of the parameter. When a language passes a parameter by reference, it passes the *address* of the parameter. If a procedure expects one type but receives the other, failure is guaranteed.

Finally, in some languages (such as PL/M) the called procedure removes the parameters from the stack before it returns to the caller. In other languages (such as C) the calling procedure removes the parameters from the stack after the called procedure returns.

In summary, items that need to be addressed for successful parameter-passing are:

- The size of each parameter.
- The order in which parameters are placed on the stack.
- The total number of parameters the procedure expects.
- Whether the parameters are passed by value or by reference.
- Whether the caller or the callee has responsibility for removing the parameters from the stack.

When mixing code from different compilers, read the documentation for both of them carefully. It should explain how each of the compilers addresses these issues.

Returned Values

The calling program expects any returned values to be located in certain pre-determined registers on return from the called function.

Almost all languages return byte and word values in the AL and AX registers, respectively. However, the registers in which pointer and doubleword values are returned varies from language to language.

Procedure Initialization and Cleanup

Some languages insert a significant amount of initialization code before the actual program begins. This code often sets up the memory environment in which the program executes. Any procedures written in that language assume that this initialization has been performed. If a main program written in another language wants to call such a procedure, that program must set up whatever environment the procedure expects.

Calls to CTOS and to the System Libraries

The following sections describe the requirements for calling CTOS library procedures. For detailed information about an individual programming language compiler, see the manual for that compiler.

Model of Computation

All CTOS library procedures use the medium model of computation. They can be called without special effort by programs compiled using any of the medium, big, large, or huge models. They can also be called by small model programs, if the program is able to explicitly specify that the call is FAR (in other words, that the called procedure resides in a different code segment).

See Chapter 10, "Stack Format and Calling Conventions" for more information about the medium model of computation.

Parameter Naming Convention

The description of the parameters to each CTOS library function follows a naming convention that identifies the function and data type of each parameter. The full naming convention is described in the preface to this manual, but the following are three important highlights:

- Any parameter whose name begins with a lowercase *p* (for pointer) is passed by reference. For example, the parameter *pFoo* is the address of *Foo*.

- Any parameter whose name does not begin with lowercase *p* is passed by value. For example, the parameter *Foo* is the value currently stored in *Foo*.

- Any parameter whose value will be changed by the called procedure has the suffix *Ret* at the end of its name. For example, the parameter *pFooRet* is the address of *Foo*, and indicates that the called procedure will change the value stored in *Foo*.

Parameter Passing Convention

CTOS calls expect parameters to be placed on the stack in order from left to right. Because the stack grows downward in memory, this means that the first parameter pushed onto the stack is at the highest address, and the last parameter pushed is at the lowest address.

In addition, all CTOS calls pop their parameters off of the stack when they return to the caller. On return from a CTOS procedure, the caller's stack pointer has been reset to the value it had before any parameters were pushed. The caller does not need to perform any stack cleanup.

See your compiler manual for information on your language's parameter-passing convention.

Returned Values

CTOS calls return values in the following registers, according to their data type.

Data Type of Returned Value	Register(s) Used
byte	AL
word	AX
doubleword	DX:AX
pointer	ES:BX

See your compiler manual for information on your language's register usage convention for returned values.

Mediation

Some compilers allow the programmer to override the defaults and specify the desired calling conventions in the source code file. Many, however, allow only the default convention for that programming language.

When the programming language convention does not match the CTOS convention, a mediator is needed. A mediator is simply a small program module which converts the native language format to the one expected by CTOS procedures.

Listing 11-1 shows an example of a mediator between a C language program and the CTOS WriteBsRecord procedure. Remember that in C, parameters are pushed in the opposite order from CTOS, and the caller (instead of the callee) pops the parameters off of the stack.

```
MedWriteBsRecord segment word 'code'
assume cs: MedWriteBsRecord
; on entry stack has:     Needs to be:
;   bp+18  sn(pcbRet)       sn(pBSWA)
;   bp+16  o(pcbRet)        o(pBSWA)
;   bp+14  cb               sn(pb)
;   bp+12  sn(pb)           o(pb)
;   bp+10  o(pb)            cb
;   bp+8   sn(pBSWA)        sn(pcbRet)
;   bp+6   o(pBSWA)         o(pcbRet)
; We need to reverse the positions of the parameters
public _MedWriteBsRecord
_MedWriteBsRecord proc far
    push bp
    mov bp,sp

    push [bp+8]    ; push pBSWA, seg and offset
    push [bp+6]

    push [bp+12]   ; push pb, seg and offset
    push [bp+10]

    push[bp+14]    ; push cb
```

continued...

```
    push [bp+18]   ; push pcbRet, seg and offset
    push [bp+16]

    callWriteBsRecord ; make the call

    mov sp,bp
    pop bp
    ret         ;don't pop original stack.  Caller does.
_MedWriteBsRecord endp
MedWriteBsRecord ends
end; Mediator.asm
```

Listing 11–1. A C-Language to CTOS Mediator

The mediator procedure (MedWriteBsRecord) accepts the same parameters as WriteBsRecord, but in the opposite order. The main work of the MedWriteBsRecord procedure consists of pushing these parameters in the order that CTOS expects them. After it has done this, MedWriteBsRecord calls WriteBsRecord.

This particular mediator does not need to pop the parameters it pushes from the stack, because the CTOS procedure pops the parameters for it. Neither does it pop the parameters it receives, because the C-language calling procedure pops them.

The mediator in Listing 11-1 only mediates a single CTOS procedure. A mediator for multiple procedures might use an Assembler macro that defines the number and size of parameters for each CTOS procedure. When a particular CTOS procedure is called, the mediator would then act according to the values set by the macro, as defined for that procedure.

See your compiler manual for information on whether your compiler supports the CTOS calling conventions natively or through a mediator.

Calls Between Languages

The requirements for successful calls between languages are similar to the requirements for calls to CTOS procedures. Some additional concerns are described below.

Model of Computation

In theory, a module written in one model of computation can call a module written in any other model of computation. To accomplish this, however, the programmer must compensate for any differences in stack format between models. In addition, some compilers may make assumptions about register contents based on the model of computation.

For example, a compiler might assume DS and SS to be equal, and use that assumption when addressing a variable. If DS and SS are in fact not equal because the procedure was called by a program compiled in a different model, problems ensue.

Only those with a solid understanding of the various models, and of their effects on code and data segmentation, should attempt to mix models.

Parameters and Returned Values

When calling between languages, the programmer must ensure that all parameter passing, stack format, and returned value conventions match. If they don't, the programmer may need to write a mediator to compensate for the differences.

Run-Time Initialization

Whenever calling a procedure that was written in a different language, a program must ensure that it creates whatever environment the procedure expects.

Floating-Point Number Formats

Most compilers use a standard IEEE floating-point number format, which is also the one used by the Intel 80x87 Numeric Coprocessors.

However, some languages (such as Basic) use a different format, and some (such as COBOL) lack floating-point numbers entirely.

If you want to mix languages that have different floating-point number formats, you can either avoid using floating-point numbers, or you can write a conversion utility for them.

12

Writing Multi-Process Programs

This chapter describes how to write a program that executes more than one process at a time. It also gives some reasons why a program might want to operate that way.

Why Use Multiple Processes?

Applications often need to do multiple things at the same time. For example, the Executive maintains a time-of-day clock in a corner of the screen while it processes user commands.

As another example, a program may need to perform work while it waits for the user's next keystroke. A single-process program could do this by repeatedly polling the keyboard, instead of actually waiting for a keystroke. It could then perform any other needed work during the time between its keyboard polls.

However, in a multi-tasking environment like CTOS this method is highly undesirable. Using a continuous polling method prevents all lower-priority tasks from running while the polling program is in its "busy-loop," checking for a keystroke and doing work in between. Even if the polling program has no work to do, it still repeatedly checks for a keystroke, wasting processor time and degrading system performance.

The solution to this type of problem is to use multiple processes.

To display the time of day, the Executive creates a second, higher-priority process which is inactive most of the time. This process gets the current time, displays it, then tells the operating system to reactivate the process after a certain period of time (approximately one minute). When the operating system reactivates it, the process performs the same functions again.

In the meantime, the main process of the Executive can receive and execute user commands. The Executive waits for keystrokes entered by the user, and processes them as they are received. Because a good deal of computer time elapses between keystrokes, the clock display process can update the clock regularly without interfering with the interactive user process.

If a program needs to perform work in between keystrokes, it could create a second, lower-priority process which performs that work. The program would then consist of two processes. The first is a normal-priority process, which waits for a keystroke and, when one arrives, performs the work associated with it.

The second process is a lower-priority one which performs the other, "background" work when there is any to be done. Giving the second process a lower priority ensures that all keystrokes are processed promptly. If the second process had a higher priority, the keyboard process would have to wait until the second process finished all its work before a keystroke could be read.

Process Management

The examples in the previous section describe two occasions where a program can take advantage of the multiprocessing capabilities of CTOS. This section describes multiprocessing in a more general way.

The Multi-Process Model

The reason for multiprocessing is efficiency. Most processes perform bursts of CPU-intensive work, each followed by a waiting period while the process accesses an I/O device. A simple example is a process which processes the records in a file. The process reads a record and performs work on it, then reads another record and performs work on it, and so on.

If the operating system allowed only one process to exist, the CPU would sit idle while the I/O device performed its work. Multiprocessing eliminates most of this idle time by switching control of the CPU between processes. Figure 12-1 shows the difference in how programs execute in a single-process model and in a multi-process model.

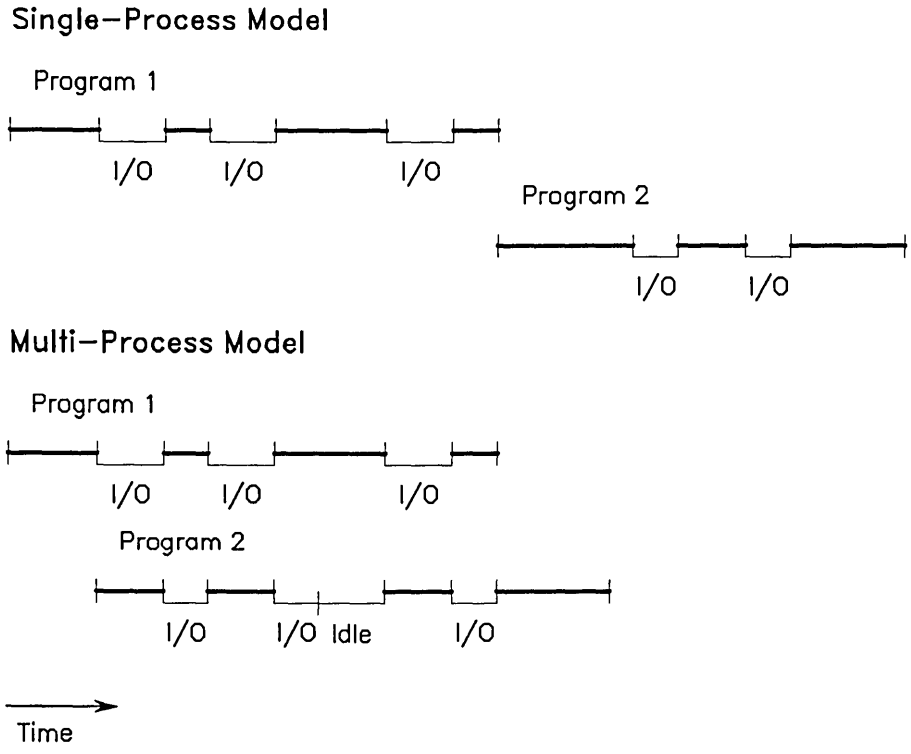


Figure 12–1. A Single-Process Model vs. a Multi-Process Model

Process States

In a multi-process environment, each process can be in one of several states at any given time. Three general process states exist.

- First, a process may be in Wait state; waiting for some external event such as I/O.
- Second, it may be in Scheduled state; ready to run, but not yet running.

- Third, it may be in Active state; running, in which case it is the process that currently controls the CPU.

Only one process runs at a time. Any number may be scheduled or waiting for some type of event.

Figure 12-2 shows the flow of a process from one state to the next. Each process in the system passes from the Active state, to Waiting, to Scheduled, to Active again, as it processes information and performs I/O.

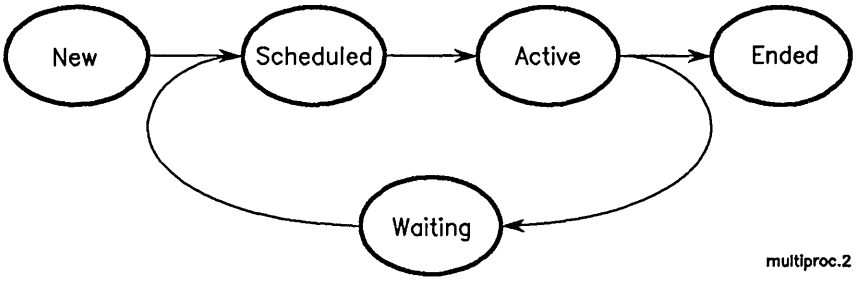


Figure 12–2. Process States

Process Scheduling

As mentioned above, only one process can control the CPU at a time. Therefore, some method is needed to allocate control to each process in a reasonable fashion. No process should be allowed to use all the available CPU time, and no process should be allowed to languish without any CPU time. Various methods have been devised to accomplish this, but this section concentrates on the one used by CTOS.

CTOS uses a method called *event-driven preemptive priority scheduling* to manage control of the CPU. This means simply that the operating system always gives control to whatever process is ready to run and has the highest priority.

The term event-driven means that whenever a system event occurs, the kernel evaluates which process should have control of the CPU. System events occur frequently, and serve many functions. For example, a

system event occurs whenever a program sends a message or issues a request, and whenever an I/O device issues an interrupt.

The term preemptive means that when the kernel evaluates which process should have control, it doesn't care whether the current process has finished its work. A process may be interrupted at any time, whenever a system event occurs. If another process with a higher priority has become ready to run, the kernel preempts the current one and transfers control to the higher priority process.

The term priority means that the kernel schedules each process based on its priority. The priority of a process indicates the importance of that process, relative to the others in the system. For example, the processes that make up the operating system have higher priority than the ones in user applications. Process priorities are discussed further in "Setting Priority" later in this chapter.

Note that if no system events occur, the kernel never gets a chance to reschedule the processes. If enough time passes between system events, a time-critical process in the operating system will be unable to execute and the workstation will crash. Therefore, programs should avoid CPU-intensive constructs (such as infinite loops) that could indefinitely prevent any system events from occurring.

The Process Control Block

The operating system stores the general information about a process in a structure called a process control block. This structure contains operating-system-specific information about the process. This information includes the priority, the default response exchange, and the user number of the process, along with other information.

Whenever a process loses control of the CPU, the processor hardware itself saves the information needed to restore the execution of that process in a Task State Segment. This information includes the process's current register contents, the current value of its instruction pointer and stack pointer, and any other information that may be needed to resume its execution.

Only the most sophisticated of system programs (such as the operating system itself) need to concern themselves with process control blocks and

Task State Segments. For the purposes of most programs, task switching is entirely transparent.

Summary

Each program can be considered a miniature copy of the system as a whole. When a program makes use of multiple processes, the same rules apply within the program that apply to the program's interaction with the system as a whole.

If the processes within a program have different priorities, those with higher priorities will always preempt those with lower priorities. This fact can be used to ensure that expected results are always achieved, as in the example where the keyboard-handling process had a higher priority.

Creating a Process

When your program starts, it consists of a single process. If you want it to have multiple processes, you must create each additional process you want. A process that creates another process is considered the *parent* of the new process. The new process is the *child* of the process that created it.

The Process Descriptor

In order to create a child process, the parent must give the operating system some information about the desired initial state for this child process. The parent accomplishes this by building a *process descriptor*. It then passes the process descriptor to the operating system as a parameter to its process-creation operation call.

The operating system provides two process-creating operations: CreateProcess and NewProcess. The CreateProcess operation uses a short form of the process descriptor, and is more convenient for most applications. The NewProcess operation uses a more complex process descriptor, allowing greater control of the child process's initial state. NewProcess also returns the process ID of the child, for later use by the parent.

Table 12-1 describes the fields in the process descriptor for the CreateProcess operation. The process descriptor for the NewProcess operation contains additional fields for the remaining processor registers. See the *CTOS/Open Application Programming Interface Specification* for more information on the NewProcess operation.

Table 12-1. Process Descriptor Fields for the CreateProcess Operation

(Page 1 of 2)

Offset	Field	Size (Bytes)	Description
0	pEntry	4	The memory address at which to begin execution of the new process. This address is usually the address of a function or procedure.
4	saData	2	The base address of the segment which contains the new process's global data. This value is loaded into the Data Segment (DS) register when the process is scheduled for execution. In all except the Large model of computation, this value must equal saStack (below). Also, if the child process needs to access data in the parent's data space, this value must equal the value of the parent's DGroup.
6	saExtra	2	The value to be loaded into the Extra Segment (ES) register when the new process is scheduled for execution. This value must be a valid segment base address or zero. It is usually zero.
8	saStack	2	The base address of the segment which contains the new process's stack. This value is loaded into the Stack Segment (SS) register when the process is scheduled for execution. In all except the Large model of computation, this value must equal saData.

Table 12-1. Process Descriptor Fields for the CreateProcess Operation

(Page 2 of 2)

Offset	Field	Size (Bytes)	Description
10	oStackInit	2	The initial value of the created process's Stack Pointer (SP) register. This value represents the top of the created process's stack, and must be a valid offset from the segment base address in saStack. oStackInit should be set to the offset of the last (highest in memory) word in a memory area allocated for the created process's stack.
12	priority	1	Priority at which the new process should be scheduled for execution. The highest possible priority is 0; the lowest is 254.
13	fSys	1	Always FALSE.
14	defaultRespExch	2	If the new process will make operating system calls, the parent process must allocate an exchange using the AllocExch operation and provide it here. The exchange becomes the default response exchange of the new process. To avoid possible conflict, the parent process must never use this exchange again.
16	fSuspend	1	If TRUE, the new process is not scheduled for execution. If FALSE, the process is scheduled for execution. FALSE is the normal setting.

Setting Up the Stack

Each process needs its own dedicated stack. In most models of computation, this stack should be in the same segment as the process's global data. In other words, the data segment register (DS) and the stack segment register (SS) should contain the same segment base address. See your compiler manual for more information about the models of computation that it uses.

There are two commonly used methods to make sure DS and SS refer to the same segment.

- Create a static array in the program's global data segment, and use that array as the child process's stack.

  This method creates a fixed-size stack, which is hard-coded in the program source file. If a program creates many processes, each must have a separate stack array.

- Type Yes in the DS Allocation field on the Linker command form, and use some or all of the DS-Allocated memory as the child process's stack.

  This method allows the stack for each process to be defined dynamically at run time. If a program creates many processes, it still must allocate a separate stack for each one.

Listing 12-1 shows a routine that allocates a large area of DS-Allocated memory, then decomposes it into stacks for three separate processes.

```
void AllocateArrays ()
{
Offset     raData;
Pointer    pData;
Selector   snData;
Word       cb, i;

/* get the selector for DGROUP */
pData = &raData;
snData = selectorof (pData);

/* call ExpandAreaSL to get the segment limit offset */
CheckErc (ExpandAreaSL (0, snData, &raData));

/* now expand the data segment to the size we need */
cb = STACKSZ * 6;
CheckErc (ExpandAreaSL (cb, snData, &raData));
```

continued...

```
/* set the offset for the three stacks */
for (i = 0; i < 3; i++) {
   switch (i) {
   case 0:
      selectorof (rgTimeStack) = snData;
      offsetof   (rgTimeStack) = raData;
      break;
   case 1:
      selectorof (rgProlStack) = snData;
      offsetof   (rgProlStack) = raData;
      break;
   case 2:
      selectorof (rgPro2Stack) = snData;
      offsetof   (rgPro2Stack) = raData;
      break;
      }
   /* set the next offset value */
   raData += (STACKSZ * 2);
   }
}
```

Listing 12–1. Using DS-Allocated Memory to Create a Process's Stack

Setting Priority

Each process must have a priority level associated with it. The priority level is simply a number between 0 and 255, with 0 representing the highest priority.

The ranges of priorities and their recommended uses are described in Table 12-2.

Table 12-2. Process Priority Values and Recommended Uses

Priority	Description
0-9	Operating System internal processes. Not for use by application programs or system services.
10-64	System Service processes. Available for use by system services, but generally not for use by application programs.
65-254	Application program processes. Most application processes run in the range from 112 to 128 (70h to 80h).
255	Null process. Used only by the operating system idle process.

Starting the Process

Starting a process is simply a matter of building a process descriptor, then calling CreateProcess. Listing 12-2 shows a procedure which receives a pointer to a function, some stack information and the address of an empty process descriptor from its caller, then creates a new process using those parameters. The function pointer should consist of the address of the child process's "main" routine.

```
void StartProcess(PF pProc, Pointer pStack, Word cbStack,
                  ProcDescType *ProcDesc)
{
Offset   raStack;
Selector saStack;
Word     exch;

/*allocate a default response exchange for the process*/
CheckErc (AllocExch (&exch));

/* get the selector and offset of the stack */
saStack = selectorof (pStack);
raStack = offsetof   (pStack);
```

continued...

```
/* set up the process descriptor */
ProcDesc->pEntry      = pProc;
ProcDesc->saData      = saStack;
ProcDesc->saExtra     = 0;
ProcDesc->saStack     = saStack;
ProcDesc->oStackInit  = raStack + cbStack - 2;
ProcDesc->priority    = 0x7F;
ProcDesc->fSys        = 0;
ProcDesc->defaultExch = exch;
ProcDesc->fSuspend    = 0;

/* create the process */
CheckErc (CreateProcess (ProcDesc));
}
```

Listing 12–2. Procedure to Create a Process

Process Termination

The only way to terminate a process is for the program that owns it to exit. If any process in a program calls one of the standard exit calls (Exit, ErrorExit, Chain, etc.), the operating system terminates all processes owned by that program. This method suits the most common type of multiprocess program, in which the program creates additional processes and uses them throughout its entire period of execution.

Some programs, however, may need to create a process, use it for a while, then dispose of it. In general, program designs that allocate and deallocate a large number of processes are not recommended for use with CTOS. This is because a program cannot fully terminate a process and deallocate the process's resources.

While it cannot deallocate the resources associated with a process, a program can ensure that a process never executes again. Once the process in question finishes its work, the process should call AllocExch, to obtain a new response exchange. It should then call the Wait primitive to wait at that exchange. Because it has never used that exchange to make a request, it will never receive a message there, and it will therefore never return from its Wait call. This "infinite wait" prevents the process from ever executing again.

Listing 12-3 shows a code fragment that performs this function.

```
void Proc2 ()
{
Pointer pMsgFoo;
Word    exchFoo;
    .
    .
    .
CheckErc (AllocExch (&exchFoo));
CheckErc(Wait(exchFoo, &pMsgFoo));
}
```

Listing 12–3. Permanently Suspending a Process

Communicating with Other Processes

A process can communicate with any other process by using the Request/ Response mechanism. In addition, processes that are owned by the same program can communicate using the Send message primitive.

The Send primitive allows a process to send a 4-byte message to another process. The message contents are undefined. The programmer can use them for any purpose desired.

If the communicating processes do not share a global data segment, it is generally unwise to use the Send primitive to pass pointers. The processes should instead use the Request mechanism (described in Chapter 1, "CTOS Overview") for that purpose.

For detailed information about the Request, Respond, and Send primitives, see Chapter 13, "Using the Kernel Message Primitives."

13

Using the Kernel Message Primitives

This chapter explains the CTOS kernel message primitives, and how using them can help to maximize the efficiency of your program.

This chapter assumes that you are familiar with the concepts of messages, requests, and exchanges. If you need more information on those concepts, see Chapter 1, "CTOS Overview" in Part I of this manual, or see your operating system documentation.

Types of Kernel Primitives

The CTOS kernel primitives fall into two basic categories: those used to send messages and those used to receive messages. The following table gives a brief description of each kernel primitive. The kernel primitives are discussed individually, in detail, later in this chapter.

Table 13-1. CTOS Kernel Primitives and Their Uses

(Page 1 of 2)

Kernel Primitive	Description
Send	Sends a four-byte message to a specified exchange. Should only be used within a program, not between programs or between partitions. No network routing.
PSend	Identical to Send, except intended for use by interrupt handlers.

Table 13-1. CTOS Kernel Primitives and Their Uses

(Page 2 of 2)

Kernel Primitive	Description
Request	Sends a predefined request block to a system service. Messages sent by this primitive can be routed over a network. Requests made using this primitive can be filtered.
ForwardRequest	Sends a predefined request block from a filter service directly to the filtered system service. Returns to the client application, bypassing the filter service. Used by one-way filters. Messages sent by this primitive can be routed over a network.
RequestDirect	Sends a predefined request block from a filter service directly to the filtered system service. Returns to the filter service. Used by two-way filters. Messages sent by this primitive can be routed over a network.
Respond	Sends a response to a Request primitive. Used by system services to respond to client applications. Messages sent by this primitive can be routed over a network.
Wait	Waits at a specified exchange for a message. Does not return until a message is posted at the exchange. This primitive suspends execution of the calling process until a message is received.
Check	Checks whether a message has been posted at a specified exchange. Returns a message if one is available, otherwise returns an error code. This primitive does not suspend execution of the calling process.

Why Use Kernel Messaging Primitives?

Using the CTOS kernel messaging primitives directly instead of using a high-level procedural interface requires significantly more work. Why bother?

Most of the time, using the kernel primitives is more trouble than it's worth. However, in certain situations using the kernel primitives can dramatically improve your application's performance. The following list describes some things you can do with the kernel primitives:

- Send asynchronous (non-blocking) requests to the operating system or to a system service.

  When a program uses the procedural interface to an operating system request or a system service request, the program halts until it receives the response to that request. In effect, it performs a Request immediately followed by a Wait. Using kernel primitives, the program can send its request, and continue to execute. Instead of waiting for the response, the program checks periodically to see if the response has arrived.

- Overlap computation and I/O operations for a process.

  Overlapping computation and I/O is really an application of asynchronous request use, but bears separate mention. Some I/O functions, such as disk operations, provide an asynchronous procedural interface. Most, however, do not. To overlap computation and I/O on devices other than disks, the program must use the kernel primitives.

- Implement customized communication between the processes in a program.

  The various processes in a multiprocess program often depend on each other to perform various functions. The kernel messaging primitives can be used for process synchronization, as semaphores, and to exchange small items of data within a program.

The following sections describe each of the kernel message primitives, and their uses. Several examples follow those sections.

Sending Messages

Two primary mechanisms exist for sending messages, the Send primitive and the Request and Response primitives. The Request primitive also has two variants, ForwardRequest and RequestDirect.

The Send Primitive

Send is the simplest of the message-sending primitives. It simply passes a four-byte message to another process. This four-byte message may itself be the message data, or it may be a pointer to a larger area which contains the message data.

See "Message-Based Operation" in Chapter 1 of this manual for general information about message passing.

The Send primitive should be used only for communication between processes within a single program. It should not be used for communication between programs or between partitions. Also, the Send primitive does not work across a network. Only local processes can communicate using Send.

There are two reasons for this. First, it is quite difficult for one program to discover another program's exchange without implementing a request to do so, and Send requires an exchange as one of its parameters.

Second, the message passed by Send is undefined, and the operating system performs no processing on it. Therefore, if one program passes a pointer to another program, that pointer is almost certainly invalid for the program that receives it, because the two programs use different Local Descriptor Tables.

The PSend Primitive

The PSend primitive is almost identical to the Send primitive. It performs the same function, but is intended for use by interrupt handlers. If an interrupt handler needs to send a message to a process, it must use the PSend primitive instead of Send.

The Request and Respond Primitives

The Request and Respond primitives are discussed in "The Request/Response Model" in Chapter 1 of this manual.

To recap that section, a client uses the Request primitive to send a request block to a system service. When the system service receives the request block, it performs the work requested, then uses the Respond primitive to send the request block back to the client.

The Request and Respond primitives are designed for a server-client model of communication, and are optimized to perform that type of communication efficiently locally, within a cluster, and over a network.

The ForwardRequest and RequestDirect Primitives

These two variations on the Request primitive are used primarily by system services which act as filters for another service's requests. For more information on filters, see "System Services that Act as Filters" in Chapter 8.

Figure 13-1 (on the following page) illustrates the use of the ForwardRequest and RequestDirect primitives.

The ForwardRequest primitive allows one program to examine a request from another program, then pass the request on to a third program. The response from the third program goes to the original requestor, not to the program that issued the ForwardRequest.

ForwardRequest is generally used by one-way filters, so that the request they have intercepted returns directly to the application that originally generated it.

The RequestDirect primitive is similar to the ForwardRequest primitive, except that the response returns to the filter program, instead of to the original requestor.

RequestDirect is generally used by two-way filters, so that the request they have intercepted returns to them for further processing. The two-way filter then uses Return to pass the request back to the application that originally generated it.

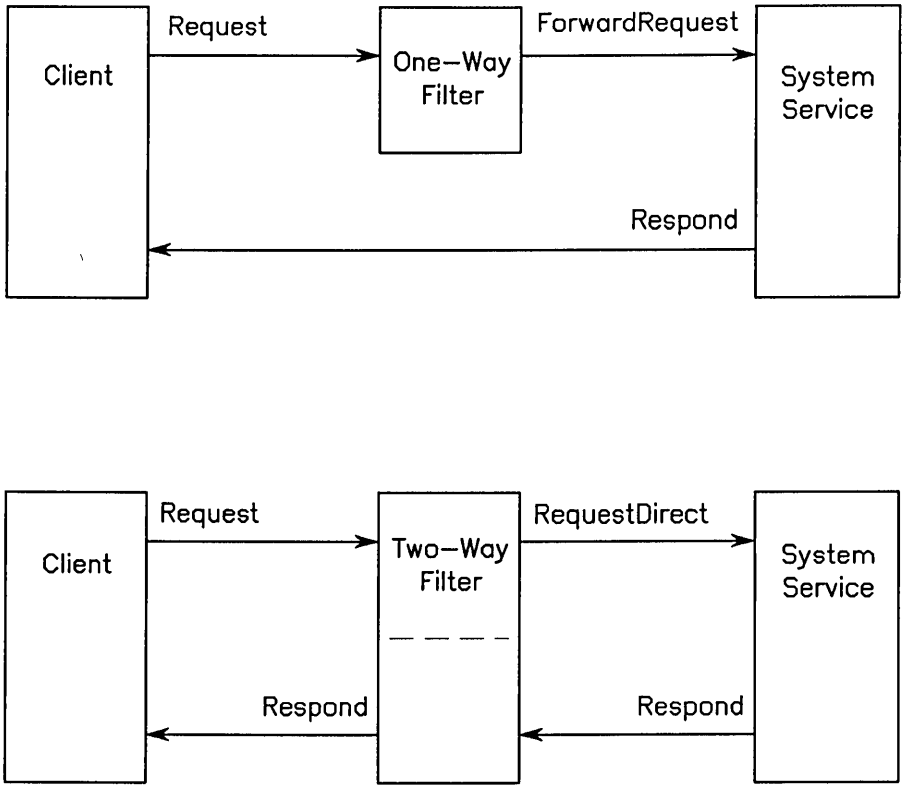


Figure 13–1. Model for Use of the ForwardRequest and RequestDirect Primitives

Receiving Messages

Two primitives can be used to receive messages: Wait and Check. When a message has already arrived, both primitives operate in the same way. They differ, however, when no message is currently available.

The Wait Primitive

The Wait primitive is the simplest way to receive a message. If a program expects a message, it can simply Wait at its exchange. When a message arrives, control returns to the program. In the meantime, the program halts its execution. If no message ever arrives, control never returns to the program.

For most programs the Wait primitive gives sufficient performance, because normally the program is only idle for a short time. In fact, the procedural interface to most operating system requests eventually results in a Request followed by a Wait.

The Check Primitive

The Check primitive differs from the Wait primitive in one simple way: the Check primitive returns control to the program immediately, whether or not a message is present at the exchange. This allows a program to poll for messages at an exchange, performing other work between polls. Programs that use Check instead of Wait have a more complex main loop, but they can be made more efficient than similar programs that use Wait.

To use the Check primitive, the program must allocate an exchange to use for asynchronous requests. It should not use its default response exchange for this purpose. The default response exchange should be reserved for the exclusive use of the operating system procedural interface.

See "Example of an Asynchronous Request," later in this chapter, for a sample program loop that uses the Check primitive.

Building a Request Block

Whenever a program uses the kernel Request primitive, it must build its own request block. Table 13-2 shows the general format of a request block. See Chapter 8, "Writing Request-Based System Services" for more detailed information about request blocks.

Table 13-2. A Sample Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	1	Size of Control Info
1	RtCode	1	Routing Code
2	nReqPbCb	1	# of Request Data Items
3	nRespPbCb	1	# of Response Data Items
4	userNum	2	Client's User Number
6	exchResp	2	Response Exchange
8	ercRet	2	Status Code Returned
10	rqCode	2	Request Code
12	fh	2	File Handle (control info)
14	lfa	4	Logical File Address (control info)
18	pbData	4	Pointer and Size of Data
22	cbData	2	(request data item)
24	pbCountRet	4	Returned Byte Count
28	cbCountRet	2	(response data item)

Listing 13-1 shows a generalized procedure for building a request block, taking advantage of the variable-length argument list allowed by the C language.

Listing 13-2 shows a code fragment that calls the procedure.

```
/* this procedure builds a request block.  It accepts a
 variable-length argument list, and builds the request
 block appropriately.
 The request-block parameter string types are:
 %B  a Byte type, %D  a DoubleWord type, %O  an Offset
type (same as Word), %P  a Pointer type, %S  a Selector
type, %W  a Word type
 */
ErcType BuildRequestBlock(RqHeaderType *pRq,
          Byte sCntInfo, Byte nReqPbCb, Byte nRespPbCb,
          Word exchResp, Word rqCode,
          char *pParamString, ...)
{
va_list pArgs; /* points to each arg in turn */
Word    userNum;
Pointer pRqBlockContents;
ErcType erc;

    /* get user number */
CheckErc(GetUserNumber(&userNum));
    /* fill in request block header */
pRq->sCntInfo = sCntInfo;
pRq->RtCode = 0;
pRq->nReqPbCb = nReqPbCb;
pRq->nRespPbCb = nRespPbCb;
pRq->userNum = userNum;
pRq->exchResp = exchResp;
pRq->ercRet = 0;
pRq->rqCode = rqCode;

    /* fill in variable part */
    /* set up argument list */
va_start(pArgs, pParamString);

    /* set pointer to first byte of rq block contents */
pRqBlockContents = pRq;
offsetof(pRqBlockContents) += 12;

while(*pParamString) {
    if(*pParamString != '%')
        pParamString++;
    else switch(*(++pParamString)) {
    case 'B':
        *(Byte *) pRqBlockContents = va_arg(pArgs, Byte);
        offsetof(pRqBlockContents) += sizeof(Byte);
        break;
    case 'D':
        *(DWord *) pRqBlockContents =
                   va_arg(pArgs, DWord);
```

continued...

```
        offsetof(pRqBlockContents) += sizeof(DWord);
        break;
    case 'P':
        *(Pointer *) pRqBlockContents =
                    va_arg(pArgs, Pointer);
        offsetof(pRqBlockContents) += sizeof(Pointer);
        break;
    case 'S':
        *(Selector *) pRqBlockContents =
                     va_arg(pArgs, Selector);
        offsetof(pRqBlockContents) += sizeof(Selector);
        break;
    case 'O':
    case 'W':
        *(Word *) pRqBlockContents = va_arg(pArgs, Word);
        offsetof(pRqBlockContents) += sizeof(Word);
        break;
    default:
        break;
    }
    }  /* end of while */
return(0);
}
```

Listing 13–1. A General Procedure to Build Request Blocks

```
    /* allocate enough memory for max req. block size */
if((pMyRq = malloc(MAXRQSIZE)) == NULL)
    ErrorExit(400);

    /* build the request block */
CheckErc(BuildRequestBlock(pMyRq, 2, 0, 2, exchMine,
         RQGETFOOTEXT, "%W %P %W %P %W", hConnect,
         &rgbServerData, sizeof(rgbServerData), &cbRet,
         2) );

    /* request a message from the server */
CheckErc (Request(pMyRq) );
```

Listing 13–2. A Code Fragment that Calls the Request Block Procedure

Where to Find Examples

Examples showing the use of kernel primitives can be found in various locations throughout this book. Repeating all the examples here would be redundant, so this section refers you to an appropriate section for each primitive.

Request

An example of the Request primitive is shown later in this chapter, in "Example of an Asynchronous Request." That example shows the most common use of the Request primitive; to allow a program to perform other work while a request is outstanding.

Respond

You can find an example using the Respond primitive in Chapter 8, "Writing Request-Based System Services" in the subsection titled "The Real Work." In that example, a system service uses the Respond primitive to respond to requests from its clients.

Send

You can find an example of the Send primitive later in this chapter, in "A Semaphore Using Send and Wait." The example shows a common use of Send within a program.

PSend

You can find an example using PSend in Chapter 15, "Timer Management" in the subsection titled "Writing the Timer Interrupt Handler." The example shows how an interrupt handler can send a message to a program process.

Wait

You can find an example of the Wait primitive later in this chapter, in "A Semaphore Using Send and Wait." You can also find an example using the Wait primitive in Chapter 15, "Timer Management," in the subsection titled "Processing Real-Time Clock Events."

Example of an Asynchronous Request

The following code fragment shows a common method of performing asynchronous requests. The main loop executes repeatedly until a key is pressed. On each iteration, the loop sends a request, then enters a subloop. The subloop repeatedly checks for a response, and performs some other work if the response has not yet arrived. Finally, when the response arrives, the main loop processes the response, then starts again at the top.

Note that a program should *always* check the status code returned by the Request primitive. If the Request primitive returns a non-zero status code, the request was not sent. Therefore, if the program called Wait under those circumstances, it would wait forever.

```
/* loop checking for a keyboard character, any one */
while (ReadKbdDirect (CHECKONLY, &bKey) ==
       ERCNOCHARAVAIL)
   {
   /* request a message from the service */
   CheckErc (Request(pMyRq) );

   /* do some other stuff while the rq is outstanding */
   while(Check(exchMine, &pMyRq) == ercNoMessage) {
      PutChar(0x2E); /* 0x2E == . */
      }

   /* display the message from the service */
   OutputToVid0 (&rgbServerData, cbRet);
   /* now wait awhile before continuing */
   CheckErc (Delay (DELAYRATE));
   }
```

Listing 13–3. A Program Loop Using the Check Primitive

A Semaphore Using Send and Wait

Semaphores are a simple but powerful application of the kernel primitives, Send and Wait. The architecture of CTOS messages and exchanges allows semaphores to be implemented in a very natural way.

As discussed earlier in this chapter, when a process Waits at an exchange for a message, control returns to the process only when a message is present. This makes it possible to implement a semaphore, simply by allocating an exchange and ensuring that no more than one message is ever present at that exchange.

To implement a semaphore in this way:

1. Allocate an exchange which will control the semaphore.

2. Send one message to that exchange to "prime" it. This message flags the semaphore as available. The message contents can be anything the program wants.

3. Write the procedures that depend on the semaphore.

   Each procedure should Wait on the semaphore exchange before entering its critical section. When it receives a message (indicating the semaphore is available) it owns the semaphore and can perform its work. When it finishes, it should Send the message back to the semaphore exchange, freeing the semaphore again.

   As long as all semaphore-dependent procedures Wait once, then Send once, only one message can ever be present on the semaphore exchange.

Listing 13-4 shows the setup code required to implement a semaphore. This code could reside in the program's main routine, or in an initialization routine.

Listing 13-5 shows a procedure that claims the semaphore, performs its work, then releases the semaphore.

```
/* Semaphore exchange is a global variable */
Word    MallocSemaphore;

void main(void) {
Pointer ptr;

/* Create a an exchange to be used as a semaphore */
CheckErc(AllocExch(&MallocSemaphore));
CheckErc(Send(MallocSemaphore, (Pointer)0x0l));
```

Listing 13–4. Code to Set Up a Semaphore

```
void *myMalloc(size_t size) {
Pointer p;
Pointer pMsgRet;

CheckErc(Wait(MallocSemaphore, &pMsgRet));

p = malloc(size);

CheckErc(Send(MallocSemaphore, (Pointer)0x0l));

return p;
}
```

Listing 13–5. A Procedure That Uses a Semaphore

14

Using Virtual Code Management – Overlays

Introduction

The Virtual Code Management facility (also called the Overlay Manager) allows you to run a program that is larger than the available memory in an application partition. This section presents practical guidelines on how to incorporate Virtual Code Management into your programs.

The Virtual Code Management Model

The Virtual Code Management facility allows the execution of programs whose code size exceeds the size of the partition in which they run. To achieve this, only portions of the code reside in memory at any given time; the remainder are on disk. It is the job of the Virtual Code Management facility to ensure that the portions of the code that are currently needed for execution are actually in memory.

The code in the run file of a program using the Virtual Code Management facility either is part of one of several *overlays*, or is resident. (Hereafter, a program that uses the Virtual Code Management facility is called an *overlay program*.) When the overlay program begins execution, the resident code is loaded into memory, where it remains for the duration of the program's execution. At some point in the program's execution, when a call is made to a procedure in one of the overlays, the Virtual Code Management facility reads that overlay into memory into an area of memory called the *overlay zone* so that the program can continue execution.

The Virtual Code Management facility keeps as many overlays as possible in memory at once. When another overlay that would exceed the available space is called into memory, the Virtual Code Management facility uses a least-recently-used (LRU) algorithm to determine which currently resident overlay to discard.

The Virtual Code Management facility is designed to run in both real mode and protected mode. If an application program is written following the rules for protected mode programs, a single overlay program run file can be created that will run in either real mode or protected mode. Which mode it actually runs in depends on which operating system is present. Guidelines for writing protected mode programs can be found in Chapter 5, "Protected Mode Programming Guidelines."

Internally, the Virtual Code Management facility operates quite differently in protected mode than in real mode. From an application program's point of view, however, there is no difference.

See your operating system documentation for an explanation of the theory behind Virtual Code Management and why you must use certain accepted call/return conventions with it. In addition, your operating system documentation includes a description of each data structure involved. While understanding that material is not required to use this chapter, you may find it helpful.

Overlay Capabilities and Restrictions

Number and Size of Overlays and Procedures

Your program may contain up to 255 nonresident code segments (*overlays*) of up to 64K bytes each.

The theoretical maximum number of procedures is 10,000 total for all segments. However, including this number of procedures in an overlaid program would require an entire 64K data segment devoted to the overlay management data structures.

Therefore, for Medium model programs the maximum number of procedures is somewhat less, since all the elements of DGroup (including the overlay management structures) must fit into a single 64K segment. In practice, this limitation very rarely impacts a program. The program is far more likely to fill up DGroup with stack space and/or with static data than with overlay management structures.

Types of Procedures that Can Be Overlaid

All procedures in an overlay that may be called from the resident portion of a program, or from another overlay, must be declared public. An overlay can contain non-public procedures, but they can only be called safely from within that overlay.

Most compilers either make procedures public by default or give you the option of declaring them public or private. See your compiler documentation for more information. A simple way to test whether a procedure is public or not is to enter the Debugger while the program is running, then attempt to examine the procedure using its symbolic name. If the Debugger does not recognize the symbolic name, the procedure is not public.

Procedures and Pointers to Procedures

The Virtual Code Management facility supports calls to a pointer to a procedure, as well as to the procedure itself. It is not necessary to eliminate these calls.

Supported Models

Large-model as well as medium-model programs can make use of the Virtual Code Management facility.

Restrictions on Interrupt Handling Routines

Interrupt handling routines must be in the resident code segment, and they must not call any procedure that is in an overlay. This restriction includes real interrupt handlers, such as those in communications programs, and also software pseudo-interrupt handlers, such as the Programmable Interval Timer.

Restrictions on Multiprocess Programs

In real mode, only one process of a multiprocess program can use overlays. Other processes must be entirely resident, including anything that they call.

While this restriction is relaxed in protected mode, it is usually wise to conform to it. This is especially true if you want to distribute a single run file that can execute in either mode.

Also, debugging a multiprocess program in which multiple processes use overlays can be extremely difficult. Programming errors in this type of program can result in strange and subtle effects whose cause is almost impossible to isolate. In general, using overlays with multiple processes is not recommended.

Segment Types that Can Be Overlaid

Only code segments, not data segments, can be placed in overlays. This restriction arises historically from the 8086/80186 processors, which do not handle faulting or paging. The Virtual Code Management facility is implemented through a software design that allows it to trap CALL and RET instructions. Instructions that write to memory cannot be trapped on those processors, so data must be resident.

Supported Languages

You can use the Virtual Code Management facility with all the system's high-level languages. In some languages, certain modules of the run-time library cannot be placed in overlays. Some other restrictions also exist with individual languages. These are discussed in the appropriate language manuals.

You can also use the Virtual Code Management facility with a program written in assembly language, if you follow the guidelines described in this section.

Call/Return Conventions

The most important single restriction on overlay programs is that they must follow the system's accepted call/return conventions.

All routines in overlays must always obey the standard call/return conventions. Even routines in the resident portion of the program must follow the conventions if they call routines in overlays. Only a resident routine that never calls an overlaid routine may deviate.

If you are using a high-level language, this point is probably not of concern to you, because almost all compilers produce code that adheres to these conventions. This point does, however, explain why certain run-time library modules must be resident in some languages: the modules do not follow call/return conventions.

If you are working in assembly language, you must follow certain rules in order to obey CTOS call/return conventions. These rules are detailed in Chapter 10, "Stack Format and Calling Conventions."

The CTOS call/return conventions order the stack arguments in such a way that the Virtual Code Management Facility can locate the return address for each call.

This ability is needed for real mode programs, which otherwise might return to a swapped-out procedure. Doing so would execute whatever code had been swapped in at that address, causing interesting results. In protected mode, CALLs and RETs to swapped-out procedures each cause the processor to generate a Not Present fault, which gives control to the Overlay Manager.

Trade-Offs

Whenever the Virtual Code Management facility brings an overlay into memory, it reads that overlay from a disk. This can take a substantial amount of time: from just a few to several hundred milliseconds, depending on the situation. Therefore, you need to consider memory versus performance trade-offs when using overlays.

Because Virtual Code Management is a set of object module procedures, its code is linked into your program. Only slight overhead (less than 5K bytes) is incurred from the code of the Overlay Manager. In addition, the

facility constructs several global data structures in your program. These data structures result in 9 bytes of overhead for each procedure in your program: 5 bytes of overhead in the resident portion of your program, and 4 bytes in the overlay that contains the procedure. While this overhead is not substantial, it can make a difference in some situations.

The location of the run file in the overall system configuration should also be considered. Each overlay is read from the place where the run file was loaded. If the run file is located on a local hard disk, performance is optimal. If the run file is being read by a cluster workstation from the master, then reading an overlay takes place at cluster speed. If the run file is located on a floppy disk, then the floppy disk cannot be removed. Performance also is limited across a network. Thus an overlay program works best from a local hard disk or on a high-speed cluster.

Basic Steps in Creating an Overlay Program

It is relatively simple and quick to write or adjust a high-level language program to use overlays successfully. The basic steps are described below. Using overlays in the most efficient way, however, is more time-consuming and is the subject of "Fine-Tuning Your Overlay Program," later in this chapter.

To create a program that uses overlays:

1. Decide upon the contents of the resident portion of the program and of each overlay.

2. Determine the appropriate size for the overlay zone.

3. Modify the program initialization to allocate memory for the overlay zone and to call InitOverlays or InitLargeOverlays to initialize the overlay zone.

4. Modify the *Object modules* list for the Linker to include /o designations for the overlay modules.

5. Relink the modified program.

Contents of Overlays

The first and most important step is to decide which modules should go into the resident portion of your code and which should be in overlays. This step requires that you thoroughly understand how your program works: how often certain routines are used and in what order they call each other.

Routines that are used so often that they would almost always be in memory clearly should be placed in the resident portion of the program. Routines that are often used together should be grouped into an overlay so that they are brought into memory together and are resident at the same time. Routines that rarely or never call each other can be in separate overlays.

You may select to restructure your source files to change the grouping of your subroutines at this point, because each overlay consists of one or more entire modules. For example, if only one subroutine out of many in a module is being called repeatedly, but the others are not, it may be worthwhile to place the frequently called subroutine in a separate module, and either make that module a separate overlay or make it part of the resident code. Otherwise, all the unused routines are read into memory and take up space every time the often-used routine is called.

Some tools you can use to analyze and refine this aspect of your program are discussed under "Fine-Tuning Your Overlay Program," later in this chapter.

Size of the Overlay Zone

The overlay zone is the memory area into which your program overlays are placed as they are read from disk. The more memory you allocate for the overlay zone, the more overlays can be present in memory at once. A small overlay zone causes overlays to be read into memory more frequently, thereby decreasing program performance.

In real mode, the overlay zone must be, at minimum, the size of the largest overlay in your program plus overhead of about 1024 bytes.

In protected mode, the overlay zone must be, at minimum, the combined size of the two largest overlays. You can approximate the overlay zone

size by considering which overlays are likely to be resident at the same time and allowing enough memory for them in the overlay zone.

In addition, if you use any very small overlays, you should note that the Overlay Manager reads from disk no less than 512 bytes at a time, and starts reading only on a 512-byte sector boundary. Therefore, a small overlay may require up to 512 bytes of additional "headroom" in the overlay zone while it is loaded.

Lastly, when you actually run the overlay program, you may find that its dynamic behavior is somewhat different from what you expected. You may want to adjust the size of the overlay zone at that point.

Tools for analysis of your program's behavior are discussed in "Fine-Tuning Your Overlay Program," later in this chapter. A method for creating a variable-sized overlay zone also is covered there.

Initialization

In most of the high-level languages, you just need to add a few lines of code early in the initialization of the program. This code must be in the resident part of the program. Typically, a program calls AllocMemorySL to allocate memory for the overlay zone, but it could statically declare a large array. It then calls one of the two procedures, InitOverlays or InitLargeOverlays, passing a pointer to the overlay zone and the desired overlay zone size.

InitOverlays takes a count of bytes as the size of the overlay zone. InitLargeOverlays is identical, except that it takes a count of 16-byte paragraphs and therefore can describe an overlay zone of up to 1M bytes.

Listing 14-1 shows a code fragment that initializes the overlay zone for a program.

```
/* Determine the size of our overlays */
CheckErc (GetOvlyStats (&OverlayStats, 6));

/* convert bytes to paragraph count */
cSwapParas = (OverlayStats.cbOvMax +
              OverlayStats.cbOv2ndMax) >> 4;
```

continued...

```
/* round it up to a 512-byte disk sector boundary */
cSwapParas = ((cSwapParas + 0x001F) & 0xFFE0);
/* add an extra sector */
cSwapParas += 0x0020;

/* Allocate memory for the Swap Buffer;  Swap buffer */
/* needs to be large enough to accomodate both the */
/* largest and second largest overlays. */
CheckErc (AllocMemorySL (cSwapParas*16, &pSwapBuffer));

/* Initialize the Virtual Code Management facility */
CheckErc (InitLargeOverlays (pSwapBuffer, cSwapParas));
```

Listing 14–1. Initializing Overlays

Linking an Overlay Program

You specify which modules are to go into which overlays when you link your program. On the *Object modules* line of the Linker form, resident object modules are named first. An object module that is to be placed in an overlay is identified by */o* appended to its name. That module and each succeeding one without a */o* are placed in one overlay.

The following example illustrates the form.

```
Bind
   Object modules    R1.obj R2.obj Ov1A.obj/o Ov1B.obj Ov2.obj/o
      .
      .
      .
```

In this example, *R1.obj* and *R2.obj* are placed in the resident portion of the program. *Ov1A.obj* is the first module in a nonresident overlay, and *Ov1B.obj* is placed in the same overlay. *Ov2.obj* is placed in a second overlay, because the */o* appended to it indicates the beginning of a new overlay.

Using this syntax, you can group object modules into overlays according to your decisions as to what is most efficient.

The Linker's default is to place all modules extracted from libraries in the resident part of the code. However, the same */o* syntax can be applied to

modules from libraries. On the *Object modules* line of the Linker form, put the name of the library and, in parentheses, the names of those library modules that you want to have placed in overlays, with */o* appended as before.

The following example illustrates the form.

```
Bind
   Object modules    R1.obj Foo.lib (Mod1/o Mod2/o)
      .
      .
      .
```

In this example, modules *Mod1* and *Mod2* from *Foo.lib* are placed in separate overlays. If the */o* had appeared only after Mod1, both modules would have been placed in the same overlay.

Note that when you want to place library modules, you must enter the names of those modules on the *Object modules* line of the command form. You cannot use the */o* designation on the *Libraries* line of the Linker form.

Remember that the total size of all the code in each overlay must be less than 64K bytes, including Virtual Code Management facility overhead.

Fine-Tuning Your Overlay Program

As mentioned earlier, it is not difficult to create an overlay program. If you want to maximize the efficiency of the overlay program's execution, you may want to use some of the tools and methods described in this section.

Overlay Zone Size

Although you can simply choose a static size for your overlay zone, it is a better practice to have your program determine dynamically what the overlay zone size should be, according to some formula. During program development and maintenance, the sizes of overlays may change. Also, an adaptable program can run in memories of different sizes on different workstations.

Your formula might, for example, set minimum and maximum overlay zone sizes. The minimum might be the sum of your two largest overlays, and the maximum, equal to the sum of the sizes of all overlays. The program can then set the overlay zone size to the appropriate value for the program's current environment.

You can get the information needed for such a formula by a call to the GetOvlyStats procedure. GetOvlyStats returns a data structure which contains the size of the largest and second-largest overlays in your program, as well as the total size of all overlays. This call is used in Listing 14-1, earlier in this chapter.

Other Memory Management Tools

ReInitOverlays and ReInitLargeOverlays

If the overlay zone is only one of several entities competing for the program's memory space (for example, if the overlay zone and a growing heap share the program's dynamically allocated memory), you may want to resize the overlay zone during execution. The two procedures for accomplishing this are ReInitOverlays and ReInitLargeOverlays.

These procedures allow the program to change the size of the overlay zone dynamically, in response to changing conditions. When you resize the overlay zone, you change the value of the highest memory address it will use. The overlay zone grows toward higher addresses, and shrinks toward lower addresses.

Shrinking the size of the overlay zone is straightforward. Simply call ReInitOverlays or ReInitLargeOverlays with the new overlay zone size. This frees the unused space at the high-memory end of the overlay zone for use by other parts of the program. It is then up to the program to identify the freed area, and use it as needed.

Enlarging the overlay zone can be somewhat more complex, because the overlay zone must remain contiguous. Before calling ReInitOverlays or ReInitLargeOverlays, the program must "clean up" the area into which the overlay zone will grow. If the program leaves valid data in the new overlay zone area, that data will eventually overwrite (or be overwritten by) an overlay, with catastrophic results.

MoveOverlays

In some cases, the cleanup required to enlarge an overlay zone cannot be accomplished without moving the overlay zone. You can relocate the overlay zone to any location, using the MoveOverlays operation. MoveOverlays simply relocates the existing overlay zone to a new place in memory.

Before using MoveOverlays, the program must ensure that the new region of memory is large enough to hold the entire overlay zone.

Performance Optimization Tools

Permanence and Release From Permanence

You can make an overlay reside permanently in memory by calling either the MakePermanent or MakePermanentP procedure. After one of these procedures is called, the specified overlay cannot be discarded until the program calls the ReleasePermanence procedure. (The overlay can, however, be moved by the MoveOverlays procedure.)

MakePermanent causes the overlay from which it is called to become permanently resident in memory. This procedure allows any overlay to make itself permanent.

MakePermanentP allows any part of the program (such as the resident part, or another overlay) to make a specified overlay permanent. To use MakePermanentP, the program passes a pointer to the overlay that it wants to make permanent. The program can get this information by calling MapIOvlyCs, which is discussed in "Utilities," later in this section. MapIOvlyCs receives an overlay number and returns the code segment (CS) of the overlay.

The program can then build a pointer, using the returned code segment and an offset of zero, which it passes to MakePermanentP. Note that all pointers a program passes to MakePermanentP should have an offset of zero.

ReleasePermanence causes all permanent overlays to revert to their normal, non-permanent status.

Changing LRU Time Stamping

The Overlay Manager's least-recently-used (LRU) algorithm chooses which overlays to discard based on their age, which is the length of time since they were read into memory. However, the Overlay Manager has no way to know how often or how recently the procedures in the overlay were called. It only knows how recently the overlay that contains them was loaded.

So, even if an overlay has the largest age value, there may have been recent activity in that resident overlay of which the Overlay Manager is not aware. The MakeRecentlyUsed procedure allows the program to correct the Overlay Manager's age information.

A call to MakeRecentlyUsed adjusts the apparent age of the overlay from which it is called, making that overlay less likely to be discarded. UpdateOverlayLRU allows any part of a program to adjust the age of any specified overlay.

Optimizing Overlay Contents

Deciding which modules to place in overlays requires you to understand what is happening dynamically in your program. For a moderate-size program, thinking carefully about how your program executes is usually sufficient. For a complex program, however, tools are available to help you analyze your overlay program's behavior.

Before using overlays at all, you should link your program and study the map file produced by the Linker. You may find that the overhead of the Virtual Code Management facility is not worth the savings in overlaying your code.

Overlaying Library Modules

Often a program may consist of a small portion of code written by the programmer and a much larger portion made up of code from libraries such as CTOS.lib, Graphics.lib, and so on. Most library routines, with the exception of communications routines and others involving interrupts, can be placed in overlays. The problem is to identify which ones are suitable candidates without knowing their internals.

To determine the module name in which a particular library procedure resides, examine the cross-reference file for that library. You can generate cross-reference files with the Librarian.

Once you know the module name, you can search the cross-reference file for the names of the other procedures in that module. Once you have that information, you can determine whether the module is a good candidate for an overlay, based on how often you call the procedures in it. For example, if you only call one of the procedures in the module, and you call that procedure only rarely, the module is a very good candidate for an overlay.

NOTE: For details on the procedures discussed in this section, see the CTOS/Open Application Programming Interface Specification.

Utilities

The following are some other overlay procedures which you can use in your overlay programs.

MapCsIOvly

You can call the MapCsIOvly procedure, passing it the code segment portion of a memory address. MapCsIOvly returns the overlay number in which that address is located.

MapIOvlyCs

Given an overlay number, MapIOvlyCs returns the code segment portion of the address in memory where it is located. Use this procedure to get an overlay's address before calling MakePermanentP or any other procedure that requires the address as a parameter.

Forcing an Overlay into Memory

You can force an overlay to be read into memory without calling a procedure in it by using MapCsIOvly and MapIOvlyCs. You can only do

this after the overlay has been in memory at least once, though. The paragraphs below explain how.

In order to force an overlay into memory, you must know its overlay number. A program can determine this at run time by using the MapCsIOvly procedure while the overlay is resident. To use this procedure, the program must first determine the code segment (CS) of one of the procedures in the overlay. See your language compiler documentation for information on how to get this value in your programming language.

Once the program knows the code segment of a procedure in the overlay, it should call MapCsIOvly to obtain the overlay number for that overlay. Thereafter, the program can force the overlay to become resident at any time by calling MapIOvlyCs. This procedure receives an overlay number, and returns the current code segment of that overlay if it is already resident. If the overlay is not already resident, the Overlay Manager loads it from disk and then returns its code segment value.

Overlay Programs in Assembly Language

While high-level language compilers shield the programmer from the pitfalls of making sure that code conforms to various conventions, the assembly language programmer has no such protection. If you are in this position, be sure to read the description of how the Virtual Code Management facility works in your operating system documentation. Also refer to your assembly language manual for rules you must follow to make sure that your overlay program will work.

In particular, your program must conform to the system's call/return conventions. In real mode, when an overlay is discarded, the stack is traced so that pointers to the discarded procedure can be corrected. The stack must be in the expected order when this occurs, or the program will fail in unexpected ways.

Troubleshooting

Linker Errors

Some errors during linking can be ignored and some indicate real problems.

Call/Return Violation

A link error that states that the program does not follow call/return conventions can arise if you have placed a library module that does not follow the conventions in an overlay.

Most such call/return errors are harmless, if the called procedure does not call any other procedures. However, if such a message is accompanied by other messages identifying illegal segment references, the error is fatal. If such a program is run, it fails sporadically. This type of error is a legitimate call/return violation. The Linker reports the offset within the module where the cause of the problem is located. You can use a code listing from the compiler or assembler to figure out which line of source code corresponds to this location. It is also possible to use the File Debugger on a run file with the symbol file and examine the instructions to locate your error.

See your Linker manual for details on each Linker error.

Debugging an Overlay Application

The Debugger is compatible with overlay programs. You can set breakpoints in overlays. When you examine code that is not currently in memory, the Debugger prints it within braces ({}).

Thrashing

Thrashing, or excessive and time-consuming disk input/output as a result of reading overlays into memory, indicates that you should reorganize the contents of your overlays. Procedures that call each other frequently should be in the same overlay.

Alternatively, you can make the overlay zone bigger.

Looping in MpRetAdrBNear

If your program fails by looping in the MpRetAdrBNear procedure, call/return conventions may have been violated, or the stack may have been overwritten in some way. (You can see this looping behavior by using the **Code-T** command to display a stack trace in the Debugger, or by using symbols from a symbol file and single-stepping.) Check to see that you have not either written nonconforming assembly language code or placed a forbidden high-level language module in an overlay.

This kind of failure occurs in real mode during the stack trace when an overlay is discarded and the Virtual Code Management facility does not find the expected format. The program runs successfully until discarding of the overlay occurs.

Linker Limitations

The Linker can detect many cases of call/return convention violations, but not all. Therefore, a real mode program failure may still encounter a call/return violation, even though the Linker returned no errors. This problem occurs mainly with programs written in assembly language.

15
Timer Management

This chapter describes the various timing services that CTOS makes available to programs. CTOS provides a range of timing services for different purposes.

CTOS Timing Services

Three types of timing services are available under CTOS. These types are:

- Delay-oriented operations

  This type of operation causes a process to pause, or "sleep" for a specific period of time.

- Real-time clock operations

  The real-time clock operations provide a mechanism for repetitive timing operations, in which a program must perform a task at regular intervals. CTOS real-time clock services support intervals as short as 100 ms.

- Precise interval-timing operations.

  Interval timing operations allow a program to set an "alarm clock" to go off after a specific period of time. Meanwhile, the program can perform other work. The programmable interval timer supports intervals as short as 50 microseconds.

Each of the three types lends itself to a different situation, and each is described in the following sections.

Using Delay and ShortDelay

The Delay and ShortDelay operations allow a program to pause for a specific period of time, then resume execution. The Delay operation allows the program to pause for any amount of time between 100 milliseconds (ms) and 1.8 hours, with a granularity of 100 ms. The ShortDelay operation allows the program to pause for any amount of time between 1 ms and 1 minute, with a 1 ms granularity.

The delay operations should be used when a process needs to stop all work for a specific, known period of time. An example of such a process is one which displays the current date and time at the top of the screen. The process should spend most if its time idle, and only "wake up" once or twice a minute to update its display. The process should not use any processor time while it is idle.

Listing 15-1 shows a procedure that performs this function. The procedure assumes the presence of an Executive-type screen which it can manipulate using VAM.

```
/* Time of day display from Multiproc.c */
void TimeProc ()
{
Word cSpaces, iCol;

while (TRUE) {
   /* get the system date and time, expand and then */
   /* display it */
   CheckErc (GetDateTime (&sysTime));
   CheckErc (NlsStdFormatDateTime (NULL, 0xFFFF, sysTime,
                &rgchTime, sizeof (rgchTime), &cchTime));
   CheckErc (PutFrameChars (1, nCols-cchTime, 1,
                &rgchTime, cchTime));
   /* now fill in the blank space from the path to the */
   /* time */
   iCol = strlen (rgchPath);
   cSpaces = nCols - cchTime - iCol;
   CheckErc(PutFrameChars(1,iCol,1,rgchSpaces,cSpaces));

   /* now wait awhile */
   CheckErc (Delay (10)); /* set to 10, for 1 second */
   }
}
```

Listing 15–1. A Time-of-Day Clock Display Procedure Using Delay

The procedure uses the Delay operation to ensure that it uses no processor time while it "sleeps". The procedure in the example should be its own, independent process, so that the Delay call affects only the time display. For more information on multi-process programs, see Chapter 12, "Writing Multi-Process Programs."

Using The Real-Time Clock

The real-time clock allows a program to perform repetitive timing operations, at precise intervals. The real-time clock can time intervals as short as 100 ms.

Note, however, that the *first* timed interval of a series can expire as much as 99 ms early. Subsequent intervals are precisely timed. If you are curious how this can happen, analyze the real-time clock process described in "Processing Real-Time Clock Events," later in this section.

Setting Up the Timer Request Block (TRB)

The real-time clock uses messages and exchanges to communicate with its clients. When a client application needs to use the real-time clock, it should first allocate an exchange, which it uses for subsequent communication with the real-time clock.

Next, the application builds a timer request block (TRB), which contains all the information about the timing service requested, and includes the number of the exchange the application allocated for communication with the timing services. Finally, the application calls the OpenRTClock procedure. This procedure tells the system timer to use the information in the TRB to perform timing services for the caller.

The TRB remains active until the client either calls CloseRTClock, or terminates.

Table 15-1 describes the format of the timer request block (TRB).

Listing 15-2 shows a procedure that sets up a TRB and opens the real-time clock.

Table 15-1. Timer Request Block Format

(Page 1 of 1)

Offset	Field Name	Size (bytes)	Description
0	counter	2	Number of 100 ms periods remaining before timer expires. Decremented every 100 ms.
2	counterReload	2	Value to be reloaded into counter when the timer expires (when counter reaches zero). If this field is set to zero, the counter does not repeat.
4	cEvents	2	Number of times the timer has expired before. If this field is zero, timer services sends a message to exchResp, then increments this field to 1. If this field is non-zero, timer services just increments the field by 1.
6	exchResp	2	The timer services client's response exchange.
8	ercRet	2	Status code. Not used by timer services. The client may use this field as desired.
10	rqCode	2	"Request code." Not used by timer services. If a client has multiple timers on the same exchange, each should place a unique value in this field. This allows the individual timers to identify their TRBs.

```
/* Build a timer block, then start the timer. */
Word InitTimer(TRBType *pTimerblock, Word cPeriods,
               Word rqCode)
{
Word exchTimer;

/* build the TRB */
pTimerblock->cEvents = 0;
pTimerblock->ercRet = 0;
pTimerblock->counter = cPeriods;
pTimerblock->counterReload = cPeriods;
pTimerblock->rqCode = rqCode;
```

continued...

```
/* allocate an exchange, and put it in the TRB */
CheckErc(AllocExch(&exchTimer));
pTimerblock->exchResp = exchTimer;

CheckErc(OpenRTClock(pTimerblock));
/* return the exchange for future communication with */
/* the timer */
return(exchTimer);
}
```

Listing 15–2. Building a TRB and Starting the Real-Time Clock

Processing Real-Time Clock Events

To use the real-time clock effectively, you should understand how timer services uses the TRB. Every 100 ms, the timer services are activated and perform the following sequence of operations for each active TRB.

1. Check the contents of the counter field. If it is zero, exit.
2. Decrement the counter field by one.
3. If the counter field is now zero, continue. Otherwise, exit.
4. If the cEvents field is zero, send the address of the TRB to the exchange specified in the exchResp field. The address sent is the address of the TRB itself, not the address of a copy.
5. Increment the cEvents field by one.
6. Copy the contents of the counterReload field into the counter field.
7. Exit.

The key event in this process is number 4. The real-time clock only sends a message to the client if cEvents is zero. This means the client must reset the cEvents field to zero at some point after receiving each message from the real-time clock. If the client fails to do so, it will wait forever for its next message from the real-time clock.

The cEvents field also allows the client to determine whether it has fallen behind its timer, and to take appropriate action. Before resetting the

cEvents field, the client should verify that its value is not greater than one. If the cEvents field contains a value greater that one, the client has missed cEvents-1 timer events. It is up to the client application to take any action necessary to make up for missed timer events.

Listing 15-3 shows a program main loop whose work is timed by the real-time clock. The program sets up the real-time clock, then performs some work every time the clock expires:

```
main() {
Word    iTimeCount, iMaxIterations, exchMyTimer;
Word    iIterations = 0, wMyRqCode;
TRBType Timerblock;
TRBType *pTRBRet;

wMyRqCode = 0x5457;
printf("Enter time interval for RealTime Clock (1 = 100
        msec):  ");
scanf("%d", &iTimeCount);
printf("\nEnter number of times RTC should fire:  ");
scanf("%d", &iMaxIterations);

/* start the real-time clock */
exchMyTimer = InitTimer(&Timerblock, 0, wMyRqCode);

/* Start Timer */
/* Load reload field first, in case interval is short */
Timerblock.counterReload = iTimeCount;
Timerblock.counter = iTimeCount;

/* loop forever */
while(TRUE) {
    /* wait for a message from the RTC */
    CheckErc(Wait(exchMyTimer, &pTRBRet));
    if(pTRBRet->rqCode != wMyRqCode)
        printf("Received wrong timer block from RTC.\n");
    /* do the work, then make sure we haven't missed a */
    /* tick.  If we have, loop until cEvents is 0. */
    do {
        DoMyWork();
        if(++iIterations >= iMaxIterations) {
            CheckErc(CloseRTClock(pTRBRet));
            return(0);
            }
        /* note:  If processor is slow, loop might not */
        /* keep up with timer.  Would therefore drop a */
        /* tick occasionally. */
                                                continued...
```

```
        if(pTRBRet->cEvents > 1) {
            printf("Missed a tick\n");
            }
    /* if we're up to date (cEvents == 0), go back to */
    /* the Wait */
    } while(--pTRBRet->cEvents > 0);
    }
}
```

Listing 15–3. Repetitive Timing Using the Real-Time Clock

Because the program may be unable to complete its work before the next time the clock expires, the program contains a secondary loop. This loop checks the value of cEvents. If cEvents is zero, the loop exits and waits for the next timer event. If cEvents is greater than zero, the wait would be infinite, so the loop instead performs the program's normal work, then decrements cEvents.

Because the program only leaves this secondary loop when cEvents is zero, the program can fall behind its timer and still continue working. Presumably, the program will eventually catch up to the timer.

Avoiding Timing Windows with the Real-Time Clock

Notice the While statement in the secondary loop of the program (the last line of code in Listing 15-3). This statement decrements the cEvents field in the TRB, then compares the field to zero. In theory, though, the timer might go off between the time the program decrements cEvents and the time it compares cEvents to zero.

If this happened, the program would process the same timer event twice. First, the program would process the event immediately, because the real-time clock incremented cEvents to one before the program checked the value of cEvents. Then, the program would process the event again because the real-time clock sent a message to the program when the timer expired.

Because of the way the example program is written, this event is very unlikely. The compiled machine code for the program decrements and compares in two successive machine instructions. Therefore, the timer

would have to expire precisely between those two instructions for the problem to occur.

There is a way to ensure to that this window never occurs, though. To do so, the program must perform all modification and comparison of the cEvents field as a single, atomic operation. A program can accomplish this in any of several ways. The simplest way is to disable interrupts before examining or modifying cEvents, then re-enable interrupts after the modification and comparison are complete.

A second, more complex, way to prevent the cEvents timing window is to make a copy of the cEvents field. To do so, allocate memory for the copy and set its value to zero. Then use the 80x86 processor's XCHG instruction to exchange the contents of the copy for the current value in cEvents. This sets cEvents to zero and sets the copy to the value in cEvents, using a single machine instruction. The program can now safely perform operations on the copy without affecting the real-time clock. Naturally, the program must perform this exchange operation every time it receives a message from the real-time clock, to get the new value of cEvents.

See your compiler manual for information on whether your compiler allows you to explicitly disable and enable interrupts, and on whether it allows you to force use of the XCHG instruction. If your compiler does not allow these things, you may need to write a subprocedure in assembly language. Or, you can find a way for the program to detect and recover from the effects of the timing window.

Timing a Single Interval with the Real-Time Clock

When a program needs to time a single interval, it can usually use the Delay function. Sometimes, though, the program may need to continue performing work while it waits for the interval to expire. It can accomplish this by using the real-time clock.

To time a single interval with the real-time clock, the program first sets up a timer block and opens the real-time clock. The program should initialize the *counter* and *counterReload* fields to zero. Then, when the program needs to time its event, it loads a value into the *counter* field, but leaves the *counterReload* field set to zero. This starts the real-time clock, ensuring that it will only time the interval once.

While the timer is counting, the program can perform its normal work, periodically calling Check for the timer exchange to see if it has received a message from the real-time clock.

Remember that for the first interval in a series, the real-time clock can expire as much as 100 ms early. This caveat applies whenever a program times a single interval, because each single interval is the only one in its series, making it the "first" interval.

Listing 15-4 shows a main loop that times a single interval with the real-time clock. Note that because the program only Checks the exchange, and does not Wait, the program will receive the clock's message some period of time after the message arrives at the exchange.

```
main() {
Word     iTimeCount, exchMyTimer, wMyRqCode;
ErcType  erc=0;
TRBType  Timerblock;
TRBType  *pTRBRet;

wMyRqCode = 0x5457;

/* get the interval */
printf("Enter time interval for RealTime Clock (1 = 100
        msec):  ");
scanf("%d", &iTimeCount);

/* set up the real-time clock, but don't start it */
exchMyTimer = InitTimer(&Timerblock, 0, wMyRqCode);
/* start the clock */
Timerblock.counter = iTimeCount;
/* loop until timer expires */
while(TRUE) {
    /* check for a message from the RTC */
    erc = Check(exchMyTimer, &pTRBRet);
    if(erc == ercNoMessage)
        DoMyWork();
    else if(erc == 0)
        break;
    else
        CheckErc(erc);
    }
CheckErc(CloseRTClock(pTRBRet));
CheckErc(DeallocExch(exchMyTimer));
}
```

Listing 15–4. Timing a Single Interval with the Real-Time Clock

Using the Programmable Interval Timer

The programmable interval timer allows you to set more precise timing intervals than the real-time clock. It also requires significantly more work to use. Using the programmable interval timer, you can time intervals as short as 50 microseconds. However, repetitive timing of very short intervals can cause substantial processor overhead, with corresponding performance degradation.

As with the real-time clock, there is a caveat about short intervals. Unlike the real-time clock, the programmable interval timer guarantees that a timed interval will *never* be shorter than the requested time. Because of this guarantee, the interval may be a few microseconds longer than the requested time.

Setting Up the Timer Pseudo-Interrupt Block (TPIB)

Like the real-time clock, the programmable interval timer uses a data structure to communicate with its client application. This structure is called the timer pseudo-interrupt block (TPIB). The TPIB contains the time interval requested, and the address of a user-written interrupt handler. The programmable interval timer calls this interrupt handler when the requested interval has passed.

The interrupt handler then performs an application-specific action. Usually, these interrupt handlers do little more than send a message to a process in the application.

To use the programmable interval timer, the programmer must perform three actions. First, she must include an appropriate timer interrupt handler in the program. Second, the program must create a TPIB, which contains the information needed by the programmable interval timer. Finally, the program must call SetTimerInt to establish each interval.

Table 15-2 describes the format of a TPIB.

Table 15-2. Timer Pseudo-Interrupt Block Format

(Page 1 of 1)

Offset	Field Name	Size (bytes)	Description
0	linkField1	4	Used by the operating system.
4	linkField2	4	Used by the operating system.
8	pIntHandler	4	The address of the timer interrupt handler's entry point (its initial CS:IP).
12	saData	2	Selector of the data segment used by the timer interrupt handler. This value is loaded into DS on entry to the interrupt handler.
14	cIntervals	2	The number of 50-microsecond periods that should elapse before the timer interrupt occurs.
16	pRqBlkRet	4	The memory address to which the address of this TPIB should be written when the timer interrupt occurs. This field allows the interrupt handler to examine a TPIB in a data segment other than the one specified in saData.
20	footPrint	2	Used by the operating system.
22	delta	2	Used by the operating system.
24	reserved	8	Reserved for use by the operating system.

Listing 15-5 shows a procedure that sets up a TPIB, and optionally starts the programmable interval timer. The procedure also allocates an exchange for communication with the timer interrupt handler.

Listing 15-6 shows the code which calls the procedure in Listing 15-5.

```
/* Build a timer block. Start timer if flag is set. */
Word InitPIT(TPIBType *pTimerblock, Word cPeriods,
             pFunc pFunction, Word fStart)
[
Pointer pData=&exchPublic;

/* build the TPIB */
pTimerblock->link1 = 0;
pTimerblock->link2 = 0;
pTimerblock->pIntHandler = pFunction;
pTimerblock->saData = selectorof(pData);
/* assumes DGROUP for int handler's DS */
pTimerblock->cIntervals = cPeriods;
pTimerblock->pRqBlkRet = &pTimerTPIB;
pTimerblock->footprint = 0;
pTimerblock->delta = 0;
pTimerblock->res1 = 0;
pTimerblock->res2 = 0;

/* allocate an exchange */
CheckErc(AllocExch(&exchPublic));

if(fStart != FALSE)
    CheckErc(SetTimerInt(pTimerblock));

return(exchPublic);
]
```

Listing 15–5. Building a Timer Pseudo-Interrupt Block (TPIB)

```
pTimerIntFunc = &HandleTimer;
    .
    .
    .
/* set up the Timer */
exchMyTimer = InitPIT(&Timerblock, iTimeCount,
                      pTimerIntFunc, fStartUp);
```

Listing 15–6. Calling the InitPIT TPIB-Building Procedure

Writing the Timer Interrupt Handler

The programmable interval timer transfers control to a user-written interrupt handler when the requested interval expires. This interrupt handler is of the mediated type, meaning that it can call operating system procedures. However, like any interrupt handler, it should be kept as brief as possible.

Listing 15-7 shows an example of a timer interrupt handler. In this example, the interrupt handler checks to see if the program has completed some portion of its work, as indicated by a flag. If the program has not, the interrupt handler logs the fact that a timeout occurred, then sends a message to the program's main process.

```
/* Timer int handler */
void  _far  HandleTimer()
{
ErcType erc;
/* if work is not done, indicate timeout.  Else do */
/* nothing. */
if(fWorkDone==FALSE) {
    /* Log the event */
    cTimeOuts++;
    /* notify the client */
    erc = PSend(exchPublic, (Pointer) 0x00000001);
    if(erc != 0)
        Crash(erc);
    }
}
```

Listing 15–7. A Timer Interrupt Handler

A Sample Program Loop Using the Programmable Interval Timer

Listing 15-8 shows a main loop that sets up and uses the sample timer interrupt handler. The main loop prompts the user for a time period, a loop iteration count, and a count of subroutine calls for each loop iteration. The program sets the timer at the start of each loop iteration, then calls the subroutine DoMyWork either until it reaches the subroutine count or until a timeout occurs.

After each loop iteration, the program resets the timer, and notifies the user if the timer went off between the time work was completed and the time at which the timer was reset. Although there are only a few instructions between these two events, certain input combinations can cause almost every loop iteration to encounter this timing window.

```
void main() {
Word    iTimeCount;
Word    exchMyTimer;
Word    cBytesRet = 0;
Word    fStartUp = FALSE;
long    fTimeOut;
TPIBType  Timerblock;
pFunc  pTimerIntFunc;
ErcType erc=0;

pTimerIntFunc = &HandleTimer;

printf("Enter time interval for the PIT (1 = 50
       microsec):  ");
scanf("%d", &iTimeCount);
printf("\nEnter count of times main loop should run:  ");
scanf("%d", &iMaxIterations);
printf("\nEnter number of times subroutine should be
       called:  ");
scanf("%d", &iSubIterations);

/* set up the Timer */
exchMyTimer = InitPIT(&Timerblock, iTimeCount,
                      pTimerIntFunc, fStartUp);
if(exchMyTimer == 0)
   ErrorExit(10);

/* loop as many times as we want */
while(iIterations < iMaxIterations) {
   /* start the timer */
   CheckErc(SetTimerInt(&Timerblock));
   putchar('\n');
   PutWord(iIterations, 10);
   putchar(' ');
   while(TRUE) {
      /* do the work */
      fWorkDone = DoMyWork();
      /* check for a message from the PIT */
      erc = Check(exchMyTimer, &fTimeOut);
      if(erc == 0) {
         WriteBsRecord(bsVid, "\nA timeout occurred",
                       19, &cBytesRet);
```

continued...

```
            iCount=0;
            break;
            }
        else if(erc != ercNoMessage)
            CheckErc(erc);
        if(fWorkDone == TRUE) {
        /* if timer went off after we finished the */
        /* work, we don't care */
            erc = ResetTimerInt(&Timerblock);
            if(erc == ercOK)
                break;
            /* if timer went off during timing window */
            else if(erc == ercBadQueue) {
                printf("\nTimer event in window.");
                break;
                }
            else
                FatalError(erc);
            }
        } /* end of while TRUE */
    iIterations++;
    }

printf("\nExecution Complete.");
printf("\nPerformed %d iterations.  %d timeouts
       occurred.", iIterations, cTimeOuts);
}
```

Listing 15–8. A Main Loop that Uses the PIT to Check for Timeouts

16

Memory Management

This chapter explains how to use the memory management features of CTOS.

Short-Lived and Long-Lived Memory

As described in the section on application memory organization in "CTOS Overview" in Part I, CTOS provides two types of dynamically-allocatable memory.

Short-lived memory exists only until the application terminates.

Long-lived memory continues to exist after the application terminates, either until the partition is removed, or until another application in that partition calls ResetMemoryLL.

Short-lived memory is the most commonly used, and is the type described in this chapter. Long-lived memory allocation works essentially the same way as short-lived allocation, so the information in this chapter also applies to that type.

The general rule for all memory allocation is this: the most recently allocated segment must be deallocated first. So, for example, if a program allocates three segments (A, B, and C) it must deallocate them in reverse order (C, B, then A).

This rule applies to all memory of a given type, but does not apply between types. In other words, long-lived memory is unrelated to short-lived memory. Short-lived segments must be deallocated in their order, and long-lived segments must be deallocated in their order, but the two orders are separate. The operating system does not care whether a program allocated a long-lived segment in between allocating short-lived segments.

Fixed-Size versus Variable-Size Data Segments

When a program allocates memory, it can allocate a fixed-size segment or a variable size segment. Whether the segment is of fixed or variable size, it counts as one segment allocation for deallocation purposes.

Figure 16-1 shows the difference between variable-size segments and fixed-size segments.

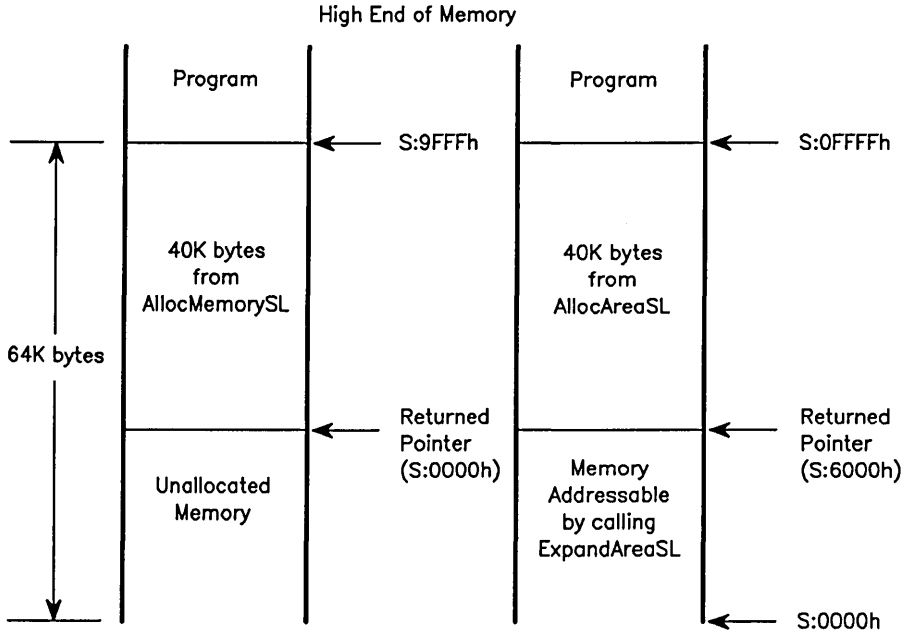


Figure 16–1. Fixed-Size versus Variable-Size Segments

Fixed-size segments are allocated by the AllocMemorySL call, and deallocated by the DeallocMemorySL call.

Fixed-size segments always start with an offset of zero. The program can then fill the segment in any way it pleases.

Variable-size segments are allocated by the AllocAreaSL call. The size of the segment can be changed by the ExpandAreaSL and ShrinkAreaSL calls. There is no explicit deallocation call for variable-length segments. To deallocate a variable-length segment, the program must use ShrinkAreaSL to shrink the segment to a length of zero bytes. The operating system then deallocates the segment.

Variable-length segments always start with a nonzero offset, and are expand-down segments. When an application allocates a variable-length segment, it receives a pointer to the bottom of the allocated area. When it expands the segment, it receives a pointer to the bottom of the expanded area. The operating system adds the new space below the existing space, but in the same segment.

Working with Variable-Size Data Segments

Variable-size data segments are created with the AllocAreaSL procedure. Once an area has been allocated, it can be expanded or contracted by the ExpandAreaSL and ShrinkAreaSL procedures.

Only the most recently allocated segment can be used as a variable-length segment. Even if a segment was allocated using the AllocAreaSL call, it cannot change size unless it is the last segment allocated.

The example in Listing 16-1 uses compatible memory management operations to allocate and deallocate short-lived memory within a segment.

Figure 16-2 shows what memory would look like just after AllocAreaSL was called in Listing 16-1. *S* represents the segment's selector. Note that a program can only reference the memory in the range of S:FC00h to S:FFFFh at this point. Attempting to access the unallocated memory shown below S:FC00h would result in a general protection fault. ExpandAreaSL expands the addressable area to lower addresses, the lowest of which in the example is S:F800h.

Any amount of allocated memory can be deallocated using ShrinkAreaSL, but the memory must be deallocated from the lowest allocated memory address.

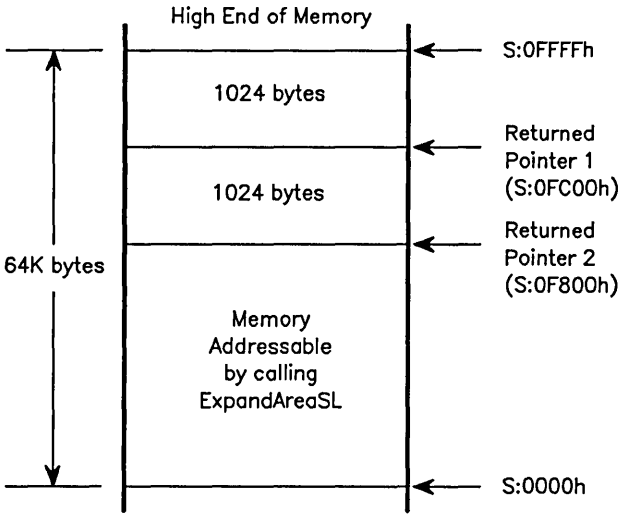


Figure 16–2. Expanding and Shrinking a Variable-Length Segment

```
void main () {
Pointer     pSegment;
Pointer     pExpandSeg;
Offset      RelAddr;
Selector    SegAddr;

/*
    Allocate an initial segment of 1024 bytes.
*/
CheckErc (AllocAreaSL (1024, &pSegment));

/*
    Need to strip the segment address (sa) for call to
    ExpandAreaSL
*/
SegAddr = selectorof (pSegment);
CheckErc (ExpandAreaSL (1024, SegAddr, &RelAddr));
```

continued...

```
/*
    Need to create pointer to last segment that was
    allocated, to give to ShrinkAreaSL.
*/
selectorof (pExpandSeg) = selectorof (pSegment);
offsetof (pExpandSeg) = RelAddr;

CheckErc (ShrinkAreaSL (pExpandSeg, 1024));

CheckErc (ShrinkAreaSL (pSegment, 1024));
/*
    Segment is now of 0 length (deallocated)
*/
}   /* End of main () */
```

Listing 16–1. Using a Variable-Length Segment

Working with Fixed-Size Data Segments

An application can allocate additional fixed-size data segments at any time, using AllocMemorySL.

As stated above, if an application needs to make many requests for small amounts of memory, it should use AllocAreaSL and ExpandAreaSL. If an application allocates large, fixed-size blocks of memory, however, it should use AllocMemorySL.

The most important fact to remember when working with multiple, dynamically-allocated segments is that segments must be deallocated in the opposite order from the one in which they were allocated. In other words, the last segment allocated must be the first segment deallocated.

Listing 16-2 shows how to allocate and to deallocate memory using AllocMemorySL and DeallocMemorySL, respectively. Figure 16-3 shows what memory would look like in the example after the second call to AllocMemorySL.

```
#define ALLOCSZ   0x2000
char  pMsglst[]= "A message in one segment concatenated";
char  pMsg2nd[]= "with a message in another segment.";

long    lgPalst, lgPa2nd;
Pointer pMemlst, pMem2nd;

void main () {

/* allocate two segments of memory */
CheckErc (AllocMemorySL(ALLOCSZ, &pMemlst));
CheckErc (AllocMemorySL(ALLOCSZ, &pMem2nd));

/* get the physical addresses */
CheckErc (PaFromP (pMemlst, 0, &lgPalst));
CheckErc (PaFromP (pMem2nd, 0, &lgPa2nd));

/* show memory, physical addresses */
printf("\nAllocated %d Byte Segment, %x:%x, at Physical
        Address %ld.", ALLOCSZ, selectorof (pMemlst),
        offsetof (pMemlst), lgPalst);
printf ("\nAllocated %d Byte Segment, %x:%x, at Physical
        Address %ld.", ALLOCSZ, selectorof (pMem2nd),
        offsetof (pMem2nd), lgPa2nd);

/* copy our messages into the allocated memory */
strcpy (pMemlst, pMsglst);
strcpy (pMem2nd, pMsg2nd);

/* display what we put in the memory segments */
printf ("\n\n%s %s", pMemlst, pMem2nd);

/* deallocate memory in opposite order of allocation   */
/* ALLFS = 0xFFFF, tells the OS to figure the size */
CheckErc (DeallocMemorySL (pMem2nd, ALLFS));
CheckErc (DeallocMemorySL (pMemlst, ALLFS));
ErrorExit (0);
}
```

Listing 16–2. Allocating Fixed-Size Segments

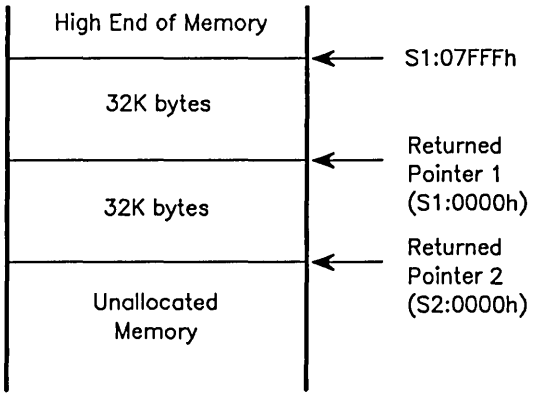


Figure 16-3. Allocating Short-Lived Memory Using AllocMemorySL

Using DS Allocation

The Linker's data segment (DS) allocation option enables run time expansion of a program's static data area. When this option is selected, the Linker assigns static data addresses at the high-order end of the data segment's 64K byte range. This leaves 64K minus the size of the program's actual static data as address space for expansion.

Figure 16-4 shows the memory image of a program that uses DS allocation.

DS allocation requires that static data be located at the low-order end of the program image. For details on DS allocation, see Chapter 10, "Stack Format and Calling Conventions," earlier in this manual.

Programs should use ExpandAreaSL to dynamically expand their static data segment. Programs that use AllocMemorySL for this purpose will not work in protected mode because, instead of expanding the existing data segment, each call to AllocMemorySL allocates a new selector for a new segment.

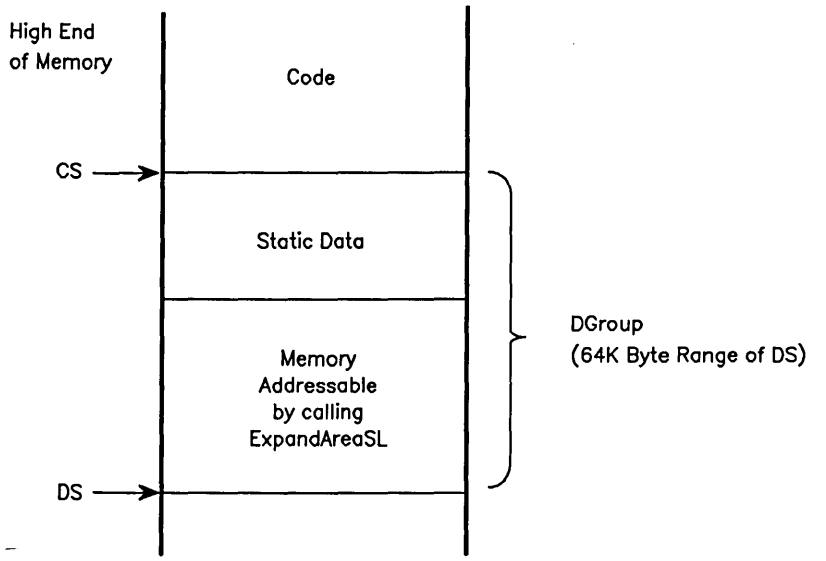


Figure 16–4. Memory Using DS Allocation

Listing 16-3 shows a program which determines the available memory in its static data segment, then allocates all of that available memory. It prints out what the highest address in that memory is, and copies a string to the new memory area.

```
#define     ALLOCSZ     0x2000
Pointer     pData;
Offset      raData;
Selector    snData;

char pStr [] = "\n\nSegment base offset in DGROUP ";

void main () {

/* we must get the selector of DGROUP */
pData = &raData;
snData = selectorof (pData);

/* call ExpandAreaSL to get the segment limit offset */
CheckErc (ExpandAreaSL (0, snData, &raData));
```

continued...

```
/* display DGROUP limit with a string in DGROUP */
printf (pStr);
printf ("before expansion is: 0%4Xh (%5u).", raData,
        raData);

/* now expand the data segment to its maximum size */
CheckErc (ExpandAreaSL (ALLOCSZ, snData, &raData));

/* display DGROUP limit with a string in expanded area */
/* of DGROUP */
offsetof (pData) = raData;
strcpy (pData, pStr);
printf (pData);
printf ("after expansion is:  0%4Xh (%5u).", raData,
        raData);

/* now shrink the segment back to the original */
CheckErc (ShrinkAreaSL (pData, ALLOCSZ));

/* call ExpandAreaSL to get the segment limit offset */
CheckErc (ExpandAreaSL (0, snData, &raData));

/* display DGROUP limit with a string in DGROUP */
printf (pStr);
printf ("after shrinkage is:  0%4Xh (%5u).", raData,
        raData);

/* get out */
ErrorExit (0);
}
```

Listing 16–3. Allocating DS-Relative Memory

Working with Data Structures Greater than 64K

The recommended method for compatible programs to build large structures is to make multiple calls to AllocMemorySL. This decomposes the large data object into multiple segments, each of which is then addressable by a returned pointer. The memory allocated by multiple calls to AllocMemorySL is contiguous.

Swapping

When the operating system swaps a program from memory to disk, it does not notify the program. When the program is swapped back into memory, nothing has changed, from the program's point of view, except the system time. For an application, being swapped is like a "lost weekend".

Applications need not do anything to prepare for, or to recover from, being swapped. System services, however, may need to know whether any of their clients have been swapped to disk. For more information on that subject, see Chapter 8, "Writing Request-Based System Services," in Part I of this manual.

17

Building a Customized SAM

Introduction

The Sequential Access Method (SAM) is a subroutine package in CTOS.lib that can be configured to include or exclude support for particular devices. It allows programs to input or output a generic stream of bytes via device-independent calls. These calls are mapped to device-dependent routines for each device type.

Three reasons to customize SAM include:

1. To include byte streams which are excluded from the default byte streams configuration.

2. To exclude byte streams from the configuration, in order to save memory at execution time.

3. To add customized byte streams for existing or new devices.

The Components of a Byte Stream

Each device-dependent portion of SAM is referred to informally as the *byte streams* for that device: for example, *video byte streams* interprets a stream of bytes sent by a program to the video device. The package of subroutines called *communications byte streams* handles input and output to and from communications ports.

A generic byte streams package is composed of

- a set of tagProcs, which identify several standard device-specific read and write operations for the device. The tagProcs operations are explained more in "Structure of SamGenAll.asm" in this chapter.

- a Byte Stream Work Area (BSWA) which is used to control operation of the byte stream. The BSWA is described in "The Byte Stream Work Area" in this chapter.

- a device name, which must be supplied by the program when it calls OpenByteStream.

- a buffer for the transferred data (described under "The Byte Stream Buffer Area" in this chapter)

The user program is responsible for providing the BSWA in its data segment. In addition, a program that uses byte streams must be linked with the correct set of byte streams. The program can then call the generic, device-independent procedures OpenByteStream, ReadBsRecord, WriteBsRecord, and so on. These generic procedures then call the appropriate device-dependent procedure for each device.

For more general information about SAM, see the description of the Sequential Access Method in Chapter 2, "Basic Input and Output."

The Default SAM Configurations

Because SAM is composed of object module routines, the byte stream routines for each device must be linked with the program and therefore make the program larger. A configuration mechanism called SamGen allows you to determine which byte streams are included, however. This lets you include the ones you use, without incurring overhead for ones you do not use.

As implemented in the SamGen.obj module in CTOS.lib, SAM is configured to include disk, keyboard, video, parallel printer, null, and spooler byte streams. This default configuration excludes communications, serial printer, generic print system (GPS), and tape byte streams.

A variant of SamGen, called SamGenAll.obj, is packaged as part of Standard Software. It includes all the bytes streams in SamGen, plus communications, GPS, and serial printer byte streams, but excludes tape byte streams. If you name [Sys]<Sys>SamGenAll.obj in the *Object Modules* field of the Linker command form, it is used in place of the default SamGen.obj.

Tape byte streams are included in CTOS.lib. To include tape byte streams, you must edit SamGenAll.asm, as described later in this section.

For details on the structure and use of communications byte streams (SamC), see Chapter 18, "Communications Programming" in this manual.

Steps to Customizing SAM

If your application uses only some of the devices defined in SamGen, you can make your executable file smaller by linking in only those byte stream packages that you need. Conversely, if you want to add a new byte streams package, you can do that as well. To accomplish this, you must create a customized SamGen module that includes only the byte streams you want, then link that module into your program. The steps required to generate a customized SamGen module are listed below. Each step is described in more detail later in this chapter.

1. Create a file, such as YourSamGen.asm, from the source file, SamGenAll.asm. (SamGenAll.asm is distributed with standard software.) Then, edit YourSamGen.asm to include only the devices your program needs to use.

2. Assemble YourSamGen.asm to produce YourSamGen.obj.

3. When you link your program, include the resulting object file, YourSamGen.obj, in the list of object modules on the Linker command form. Your module then takes precedence over the default SamGen module in CTOS.lib.

The SamGenAll.asm File

SamGenAll.asm is an assembly language source file which defines the device-dependent procedures for each device SAM supports. The file uses assembly language macros to present the definitions in a fairly easy-to-use format. The file also contains extensive comments that explain how to edit it. You do not have to know assembly language to do so.

The definitions of the macros themselves are contained in the file SamGen.mdf but you should rarely, if ever, need to examine or modify them.

This section gives a brief overview of the macros and their functions. The macros are described in more detail later in this chapter.

After initial comments, SamGenAll.asm lists several macros which use the following template:

```
%DeviceOpen ([DeviceName], OpenProc)
```

One %DeviceOpen macro appears for each type of device. It identifies the device-dependent procedure which opens that device. The procedure named on the %DeviceOpen macro is called whenever a program calls OpenByteStream for that device name.

The %DeviceOpen macros are followed by a comment explaining how to edit the next list of macros, which have the form

```
%tagProcs (tagName, FillBufferProc,
          FlushBufferProc, CheckPointBsProc,
          ReleaseProc, SetImageModeProc)
```

There is one %tagProcs macro for each mode of operation (read, write and modify) supported by each device. These macros map a device-dependent routine to each of the generic SAM procedures for that device in that mode. For example, WriteBsRecord calls the procedure defined as FlushBufferProc when the buffer for the device is full.

Finally, after another comment, a few more macros of this form appear:

```
%DevDepProc (DevDepCall, DevDepProc)
```

These macros identify device-dependent routines which can be used only with that device.

Editing the SamGenAll.asm File

Excluding Byte Streams

If an application program does not use a particular device, you can reduce the program's memory requirements by excluding that byte stream from the customized SamGenAll.obj. To do so, delete the following macros associated with the byte stream device you want to omit:

- the %DeviceOpen macro for the device
- all %tagProcs macros for the device
- any %DevDepProc macros for the device

For example, to exclude disk and spooler byte streams, you would delete the following source lines from SamGenAll.asm:

```
%DeviceOpen ([Disk], OpenByteStreamAD)
%DeviceOpen ([Spl], OpenByteStreamSpl)

%tagProcs (tagDiskRead, FillBufferAD,
          FlushBufIllegal, CheckPointBsAD,
          ReleaseByteStreamAD,
          SetImageModeIllegal)
%tagProcs (tagDiskWrite, FillBufIllegal,
          FlushBufferAD, CheckPointBsAD,
          ReleaseByteStreamAD,
          SetImageModeIllegal)
%tagProcs (tagDiskModify, FillBufferAD,
          FlushBufferAD, CheckPointBsAD,
          ReleaseByteStreamAD,
          SetImageModeIllegal)
%tagProcs (tagSplWrite, FillBufIllegal,
          FlushBufferAD, CheckPointBsAD,
          ReleaseByteStreamSpl,
          SetImageModeSpl)

%DevDepProc (GetBsLfa, GetBsLfaAsync)
%DevDepProc (SetBsLfa, SetBsLfaAsync)
```

In this example, two device-dependent procedures, GetBsLfa and SetBsLfa, are associated with disk byte streams and must be deleted.

After editing SamGenAll.asm, you must assemble it and include the resulting SamGenAll.obj in the *Object modules* field of the Linker command form when you link your program. This causes the Linker to use your customized SamGen module instead of the one in CTOS.Lib.

Including Communications, GPS, and Serial Printer Byte Streams

Serial printer and communications byte streams are not included in the default SamGen.obj module in CTOS.lib. You can add them by linking with [Sys]<Sys>SamGenAll.obj.

Adding a User Written Byte Stream

You can add your own byte stream to the SAM configuration in CTOS.lib. To do so,

1. Write the six necessary device-dependent routines, and add them to CTOS.lib:

   OpenProc, to open the device

   FillProc, to fill the device's buffer

   FlushProc, to flush the device's buffer

   CheckPointProc, to verify that a read or write has been completed

   ReleaseProc, to close the device

   SetImageModeProc, to set the device to binary or ASCII mode, if appropriate

2. Edit SamGenAll.asm to include entries for the new byte stream.

Templates for and descriptions of the six device-dependent routines are given under "Template %tagProcs Routines," later in this chapter.

For example, to add a read-only byte stream and a write-only byte stream for a device Foo, add the following entries to SamGenAll.asm:

```
%DeviceOpen ([Foo], OpenByteStreamFoo)

%tagProcs (tagFooRead, FillBufferFoo,
          FlushBufIllegal, CheckPointBsFoo,
          ReleaseByteStreamFoo,
          SetImageModeFoo)
```

```
%tagProcs (tagFooWrite, FillBufIllegal,
          FlushBufferFoo, CheckPointBsFoo,
          ReleaseByteStreamFoo,
          SetImageModeFoo)
```

If SetImageModeFoo is inappropriate for your type of byte streams, specify SetImageModeIllegal instead of SetImageModeFoo in the %tagProcs macro.

After editing, you would assemble SamGenAll.asm and include the resulting SamGenAll.obj in the list of object modules in the Linker command form.

Substituting an Alternate Byte Stream

You can also substitute an alternate version of a byte streams package instead of using the version contained in CTOS.lib. This new version can be a byte stream that came with your operating system or one that you have written.

For example, to substitute synchronous disk byte streams for the default asynchronous disk byte streams, you should replace the first group of source lines shown below with the second group:

Group 1:

```
%DeviceOpen ([Disk], OpenByteStreamAD)

%tagProcs (tagDiskRead, FillBufferAD,
          FlushBufIllegal, CheckPointBsAD,
          ReleaseByteStreamAD,
          SetImageModeIllegal)
%tagProcs (tagDiskWrite, FillBufIllegal,
          FlushBufferAD, CheckPointBsAD,
          ReleaseByteStreamAD,
          SetImageModeIllegal)
%tagProcs (tagDiskModify, FillBufferAD,
          FlushBufferAD, CheckPointBsAD,
          ReleaseByteStreamAD,
          SetImageModeIllegal)
```

continued...

```
%tagProcs (tagSplWrite, FillBufIllegal,
          FlushBufferAD, CheckPointBsAD,
          ReleaseByteStreamSpl,
          SetImageModeSpl)

%DevDepProc (GetBsLfa, GetBsLfaAsync)
%DevDepProc (SetBsLfa, SetBsLfaAsync)
```

Group 2:

```
%DeviceOpen ([Disk], OpenByteStreamSD)

%tagProcs (tagDiskRead, FillBufferSD,
          FlushBufIllegal, CheckPointBsSD,
          ReleaseByteStreamSD,
          SetImageMode)
%tagProcs (tagDiskWrite, FillBufIllegal,
          FlushBufferSD, CheckPointBsSD,
          ReleaseByteStreamSD,
          SetImageMode)
%tagProcs (tagDiskModify, FillBufferSD,
          FlushBufferSD, CheckPointBsSD,
          ReleaseByteStreamSD,
          SetImageMode)
%tagProcs (tagSplWrite, FillBufIllegal,
          FlushBufferSD, CheckPointBsSD,
          ReleaseByteStreamSpl,
          SetImageMode)

%DevDepProc (GetBsLfa, GetBsLfaSync)
%DevDepProc (SetBsLfa, SetBsLfaSync)
```

Note that because spooler byte streams share several routines with disk byte streams, the %tagProcs macro for spooler byte streams must also be replaced in this example.

After editing SamGenAll.asm (or a copy of it), you must assemble the file, then include the resulting object module in your list of object modules on the Linker command form.

Adding New Byte Streams

The previous parts of this chapter have described how to modify SAM to include or exclude various existing byte streams. The following sections describe how to create your own. These byte streams may supplement existing ones, or they may replace them entirely.

Each of the following sections covers one aspect of creating a new byte stream.

The Byte Stream Work Area

The Byte Stream Work Area (BSWA) is a 130-byte memory work area for use by the various byte streams. The first 14 bytes of the BSWA are common among all the byte streams and are used by the device-independent part of SAM. The other 116 bytes are available for use by the device-dependent operations to record the internal state of the byte stream.

The 14-byte common area must be initialized by the OpenProc routine for the device. The format of the common area and the values to which each field should be initialized are shown below:

Offset	Field	Size (bytes)	Initial Value
0	pBuffer	4	address of user-supplied buffer.
4	sBuffer	2	buffer size (bytes)
6	ibRead	2	(see below)
8	ibWrite	2	(see below)
10	fOkToPutBack	1	TRUE or FALSE
11	fPutBack	1	FALSE
12	bPutBack	1	0
13	tag	1	appropriate tag value

where

pBuffer — Is the address in memory of the byte stream buffer area. This parameter is received as a parameter on the OpenByteStream call.

sBuffer	Is the size in bytes of the byte stream buffer area. This parameter is received as a parameter on the OpenByteStream call.
ibRead	Is the read position in the buffer. If the byte stream is write only, then ibRead should be set to 0FFFFh. Otherwise, ibRead should be set to 0.
ibWrite	Is the write position in the buffer. If the byte stream is read only, ibWrite should be set to 0FFFFH. Otherwise, ibWrite should be set to 0.
fOkToPutBack	Is TRUE if the byte stream supports the PutBackByte procedure. Some byte streams (disk byte streams, for example) allow you to read a byte and, if you want to, to put it back. For example, if the current character is a delimiter indicating the start of a new string, you can put it back so that when reading starts again later, you get the first byte of the new string.
fPutBack	Is TRUE if a byte was put back.
bPutBack	Is the most-recently put-back byte if fPutBack is TRUE.
tag	Is a byte that identifies the type of byte stream.

In the above table, TRUE is 0FFh and FALSE is 0.

The Byte Stream Buffer Area

The byte stream buffer area is a user-supplied memory area which is used by the various byte streams to buffer input and output. The pBuffer and sBuffer fields of the BSWA describe a buffer for the device-independent part of SAM. This buffer must be word-aligned. The device-independent part of byte streams moves user data to or from this buffer, in response to the program's SAM procedure calls. When the buffer is exhausted, a device-dependent routine is called to write or read a new buffer. Descriptions of three different buffering schemes follow.

No Buffering

Some byte streams (for example, keyboard and video) do not buffer their input or output, and therefore ignore the byte stream buffer area. These byte streams set the sBuffer field in the BSWA to zero. When a byte stream uses this method, every read or write operation causes a device-dependent routine to be called.

Single Input/Output Buffer

Some byte streams (such as synchronous disk byte streams) use a single input/output buffer. These byte streams use the pBuffer and sBuffer fields in the BSWA to describe this buffer. When a byte stream uses a single buffer, it can perform only one operation at a time.

Pool of Asynchronous Buffers

Some byte streams (such as asynchronous disk byte streams) divide the byte stream buffer area into a pool of input/output buffers which are used with asynchronous input and output. The pBuffer and sBuffer fields in the BSWA describe the next available buffer for use by the device-independent part of byte streams. Meanwhile, other buffers from the pool may be involved in asynchronous input and output. As operations continue, the pBuffer and sBuffer fields change to indicate the next available buffer.

For example, asynchronous disk byte streams divide the byte stream buffer area into two buffers (called A and B here). For an output byte stream (mode Write), buffer A may be used for an asynchronous write to the disk while buffer B is being filled with user data by the device-independent part of byte streams. At this point, pBuffer and sBuffer describe buffer B.

When buffer B becomes full, the device-dependent routine, to which we give the generic name FlushBufferProc here, is called to write buffer B. FlushBufferProc makes sure that any asynchronous write operations involving buffer A have finished and then starts an asynchronous write operation with buffer B. It then sets the pBuffer and sBuffer fields of the BSWA to point to buffer A, which either is idle or has just finished its own asynchronous write operation.

Opening a Byte Stream and the %DeviceOpen Macro

The macro %DeviceOpen (deviceName, OpenProc) declares a procedure generically referred to here as OpenProc. The OpenProc procedure is called by OpenByteStream when a program attempts to open deviceName. In other words, every time OpenByteStream is called with a device specification that matches the deviceName declared on this %DeviceOpen macro.

For example, if Samgen.asm contains the line "%DeviceOpen([Foo], OpenFooProc)", then the procedure OpenFooProc is called whenever a program attempts to open the device [Foo] by calling OpenByteStream.

OpenByteStream identifies the appropriate OpenProc by attempting to match the passed device specification with each of the defined deviceNames. If a match occurs, OpenByteStream calls the appropriate OpenProc procedure. If no match occurs, OpenByteSteam does the following:

- If the device specification is not entirely enclosed by brackets ([. . .]), and disk byte streams are included in the SAM configuration, the device specification is assumed to be a file specification, and the OpenProc for disk byte streams is called.

  If disk byte streams have not been included in the SAM configuration, OpenByteStream returns status code 7 ("not implemented") in this case.

- If the device name is in brackets, and the Generic Print System byte streams have been included, the device specification is assumed to be a generic print device.

- If the device name is in brackets, and the spooler byte streams have been included, the device specification is assumed to be a spooler queue name.

When OpenByteStream tests for a match, it only compares the number of characters contained in the device name on the %DeviceOpen macro declaration. Thus, if a program calls OpenByteStream with a device specification of [Comm]B, OpenByteStream matches the device name to the [Comm] device. It ignores the B, which is used later by the OpenProc procedure for the [Comm] device.

For more information on OpenProc procedures, see "Template %tagProcs Routines," later in this chapter.

The %tagProcs Macro

The %tagProcs macro declares a tag name and five device-dependent procedures, each of which is called by the corresponding device-independent procedure. The tagProcs macro is defined as

```
%tagProcs (tagName, FillBufferProc, FlushBufferProc,
          CheckPointBsProc, ReleaseProc,
          SetImageModeProc)
```

The tagName field should usually take the form "tag<DeviceName> <OpenMode>". For example, the %tagProcs macro for the [Foo] device opened in write mode would be "tagFooWrite".

The device-dependent procedures correspond to the SAM device-independent procedures as shown in Table 17-1, below.

Table 17-1. Correspondence of %tagProcs Procedures to Byte Stream Procedures

%tagProcs Procedure Name	Byte Streams Procedure Name
FillBufferProc	ReadByte, ReadBytes, ReadBsRecord
FlushBufferProc	WriteByte, WriteBsRecord
CheckPointProc	CheckPointBs
ReleaseProc	ReleaseByteStream
SetImageModeProc	SetImageMode

When a program calls CloseByteStream, the generic portion of SAM calls CheckPointProc, then calls ReleaseProc.

The tagName for a device is declared as a public byte variable by the macro and is given a value. This value is used to route the device-independent calls to the appropriate device-dependent routines.

Each OpenProc procedure should declare its tagName as an external byte variable, and should assign its value to the tag byte in the BSWA when it is called. This allows the OpenProc procedure to gain access at runtime to the value assigned by the %tagProc macro. (See "Opening a Byte Stream and the %DeviceOpen Macro," above.)

The procedural interfaces for the template procedures FillBufferProc, FlushBufferProc, CheckPointProc, ReleaseProc, and SetImageModeProc are given in "Template %tagProcs Routines," later in this chapter.

The %DevDepProc Macro

The macro %DevDepProc declares a device-dependent procedure which is called whenever a program calls the device-dependent procedure name specified on the macro. The syntax of the %DevDepProc macro is:

```
%DevDepProc (devDepCall, devDepProc)
```

devDepCall is the procedure name that programs can use. devDepProc is the actual name of the procedure.

This macro allows for substitution of alternate routines for any existing device-dependent byte stream routines (such as QueryVidBs, GetBsLfa, SetBsLfa, or SetImageMode).

Operating System-Supplied Null Routines

Several null routines contained in CTOS.lib may be useful if you are writing your own byte stream. These routines do nothing except return a status code.

The routines FlushBufIllegal, FillBufIllegal, and SetImageModeIllegal all return status code 7 ("Not implemented") when called. These routines can be assigned on the %tagProcs macro if that type of operation is not

supported by the device. For example, they might be used with read-only and write-only byte streams or with byte streams for which the SetImageMode procedure is inappropriate.

The routine ChkptNop returns status code 0 (normal execution) when called and can be used when CheckPoint actions are unnecessary. For example, this procedure might be used with read-only byte streams.

The routine ReleaseEasy returns status code 0 when called and can be used with devices that require no disconnect logic (such as the keyboard).

To use one of these routines for your byte stream, simply put its name in the appropriate field on the %tagProcs macro.

Error Checking

All byte stream routines should check the input parameters for errors. The following paragraphs give some general guidelines for error checking in customized byte streams.

The OpenProc routine for a device should check that the open-mode parameter is appropriate to the byte stream. If the mode is not appropriate, OpenProc should return status code 2315 ("Invalid mode").

For example, OpenByteStreamK (for the keyboard byte stream) should check that the mode parameter is equal to mode Read.

Second, each of the device-dependent routines FillBufferProc, FlushBufferProc, CheckPointProc, ReleaseProc, and SetImageModeProc should check the tag byte in the received BSWA to make sure it specifies the correct device for the device-dependent routine. If the BSWA tag is not appropriate, the device-dependent routine should return status code 2325 ("Invalid BSWA").

For example, ReleaseByteStreamD should check that the BSWA tag byte is either TagDiskRead or TagDiskWrite.

Template %tagProcs Routines

The template routines FillBufferProc, FlushBufferProc, CheckPointProc, ReleaseProc, and SetImageModeProc are described next in this section.

OpenProc

OpenProc (pBSWA, pbFilespec, cbFilespec, pbPassword, cbPassword, mode, pBufferArea, sBufferArea): ercType

Description

The OpenProc routine is called when a program calls OpenByteStream for the corresponding device. At a minimum, the OpenProc routine must perform the following functions:

1. Place appropriate values in the control fields of the Byte Stream Work Area, and set the tag byte to tagIllegal. This initializes the BSWA, without allowing the program to call it yet.
2. Check that the mode (read, write or modify) is valid for the device. Return erc 7 (not implemented) if the mode is invalid.
3. Perform any operations required to open the device itself.
4. Initialize the buffer(s), if any, used for the byte stream.
5. Place the appropriate tagName value in the tag field of the BSWA, then return. The byte stream is now open, and the program can make calls to it.

Received Parameters

pBSWA Is the memory address of the Byte Stream Work Area.

pbFilespec Is the address of the character string that identifies the device.

cbFilespec Is the length of the character string that identifies the device.

pbPassword Is the address of the password string for the device.

cbPassword Is the length of the password string.

mode	Is the open mode for the device.
pBufferArea	Is the address of a user-provided buffer area for data exchange with the device.
sBufferArea	Is the size of the user-provided buffer area.

FillBufferProc

FillBufferProc (pBSWA, cbMax, pPbRet, pCbRet): ercType

Description

The FillBufferProc routine is called whenever the program attempts a read operation with an empty buffer. An empty buffer condition is detected when the ibRead field in the BSWA is greater than the sBuffer field. The ibRead field is an index which points to the first unread byte in the buffer described by the pBuffer and sBuffer fields. The FillBufferProc may do either or both of the following when called:

- Fill the buffer and set ibRead to a value less than sBuffer. The user's subsequent read byte stream operations are served from this buffer until it is exhausted. If no additional string is to be passed back to the user (see below), set cbRet to zero.

- Pass a string of bytes (up to cbMax long) back to the user. The pbRet and cbRet parameters should be set to point at the string. If this string is the only data to be passed back to the user (no buffer), the pBuffer and sBuffer fields in the BSWA should remain unchanged.

Byte streams using buffered input/output should pass a buffer of data back by setting cbRet to zero and updating pBuffer, sBuffer, and ibRead (the first case above). The generic part of SAM then reads the data from the buffer and passes it to the program.

Byte streams using unbuffered input/output (for example, keyboard) should keep ibRead set to 0FFFFh to force a call to FillBufferProc for every read operation. FillBufferProc should then return bytes via cbRet and pbRet (the second case above).

The procedure FillBufIllegal is a fill buffer routine that returns status code 7 ("Not implemented") whenever it is called. Byte streams for which read operations are illegal should set ibRead to 0FFFFh. The %tagProcs macro for this type of byte stream should name FillBufIllegal as the FillBufferProc parameter.

Received Parameters

pBSWA Is the memory address of the same Byte Stream Work Area that was supplied to OpenByteStream.

cbMax Is the maximum count of bytes of data that the calling process will accept.

pPbRet Is the memory address of 4 bytes into which the memory address of the data is returned.

pCbRet Is the memory address of a word into which the actual count of data bytes made available is returned.

FlushBufferProc

FlushBufferProc (pBSWA, pb, cb, pCbRet): ercType

Description

The FlushBufferProc routine is called whenever the program attempts a write operation with a full buffer. The ibWrite field of the BSWA is an index pointing to the next byte of the buffer described by the pBuffer and sBuffer fields. A full buffer condition is detected when ibWrite is greater than or equal to sBuffer.

The FlushBufferProc may do either or both of the following when it is called:

- Write the contents of the buffer to the device and set ibWrite to a value less than sBuffer. The user program's subsequent write byte stream operations are serviced using this buffer until it fills up again. If no additional string is to be written (see below), set cbRet to zero.

- Write a string of bytes described by the pb and cb parameters. The cbRet word should be set to the count of bytes written from this string. If this string is the only data to be written (no buffer), the pBuffer and sBuffer fields in the BSWA should remain unchanged.

Byte streams using buffered input and output should write a buffer of data to the device, set cbRet to zero, and update pBuffer, sBuffer, and ibWrite (first case above).

Byte streams using unbuffered output (for example, video) should keep ibWrite set to 0FFFFh so that FlushBufferProc is called for every write operation. FlushBufferProc should then write the string described by cb and pb to the device, then return the number of bytes written in the cbRet parameter (second case above).

The procedure FlushBufIllegal is a flush buffer routine that returns status code 7 ("Not implemented") whenever it is called. Byte streams for which write operations are illegal should set ibWrite to 0FFFFh. The %tagProcs macro for this type of byte stream should name FlushBufIllegal as the FlushBufferProc parameter.

Received Parameters

pBSWA Is the memory address of the same Byte Stream Work Area that was supplied to OpenByteStream.

pb Is the memory address of the data to be written.

cb Is the count of bytes to write.

pCbRet Is the memory address of the word into which the count of data bytes successfully written is returned.

CheckPointProc

CheckPointProc (pBSWA, fIsPartOfClose): ercType

Description

The CheckPointProc procedure performs a checkpoint on the open output byte stream specified by the BSWA. CheckPointProc writes any partially full buffers to the device, then waits for all write operations to complete successfully before returning. The flag parameter fIsPartOfClose is TRUE if this call to CheckPointProc is part of a device-independent call to CloseByteStream.

Received Parameters

pBSWA	Is the memory address of the BSWA.
fIsPartOfClose	Is TRUE if this call to CheckPointProc is part of a call to the device-independent procedure CloseByteStream.

ReleaseProc (pBSWA): ercType

Description

The ReleaseProc procedure closes the device or file associated with the open output byte stream specified by the BSWA. ReleaseProc does not write any buffers before closing the device or file. If partially full buffers need to be written to the device, the byte stream's CheckPointProc should be called.

Typically, ReleaseProc is called when a CloseByteStream call returns a status code other than 0. ReleaseProc should relinquish any and all resources that the byte stream has reserved (for example, exchanges, files, memory, and so on), then return.

Received Parameters

pBSWA Is the memory address of the BSWA.

SetImageModeProc

SetImageModeProc (pBSWA, mode): ercType

Description

The SetImageModeProc procedure sets the normal, image, or binary mode for printer, spooler, and communications byte streams. If attempted on other byte stream types, it should return status code 7 ("Not implemented").

SetImageModeProc is used to control the transparency or interpretation of escape codes or control characters embedded in the byte stream. For example, some printer drivers use escape sequences to position the print head, and to perform other control functions. However, if you wish to print binary data, you may want to turn off escape sequence processing, because some sequences of binary bits may look like escape sequences.

Binary mode is used for raw binary data. All interpretation of data as control is turned off.

On byte streams that support various image modes, the SetImageModeProc simply sets an image-mode flag somewhere in the BSWA. The FillBuffer and FlushBuffer procs must act appropriately, based on the setting of the flag.

Received Parameters

pBSWA Is the memory address of the same BSWA that was supplied to OpenByteStream.

mode Is a code as follows:

0 for normal mode
1 for image mode
2 for binary mode

Normal mode performs character conversion on the byte stream. For example, tabs are converted to spaces and line endings are converted to the appropriate device codes.

Image mode performs no character conversion, but recognizes escape codes.

Binary mode performs no character conversion, and recognizes no escape codes. It simply passes bytes to the device.

18

Communications Programming

Layered Support for Communications Programming

CTOS provides interfaces to its communications features at several different levels. Each communications application can use the interface which gives the appropriate balance between performance and ease of programming.

The higher-level interfaces, SAM and SamC, are simple to use and deliver good performance for most asynchronous communications applications. The lower-level interface, CommLine, is less simple to use but provides an extremely high-performance interface suitable for high-speed communications, and for synchronous communications applications.

What Is SAM for Communications (SamC)?

Communications byte streams, or SamC, is the RS-232-C device-dependent portion of the Sequential Access Method (SAM). It is the standard CTOS driver for asynchronous RS-232-C serial communication. This includes the use of communication ports for terminals, modems, and serial printers, as well as for direct connection.

Using the standard CTOS RS-232-C driver frees the application programmer from writing interrupt service routines, buffer management procedures, serial controller initialization sequences, and other low-level device-oriented routines. Unless your application performs synchronous RS-232-C communication, communications byte streams should be flexible enough to meet all its serial communication needs.

You can use communications byte streams indirectly, using the normal SAM procedures, or you can take advantage of the special device-dependent subroutines of SamC. Using the generic SAM

procedures preserves device independence, but suffers a slight performance penalty. Using the SamC procedures loses device independence, but gives greater control over the serial device interface and delivers better performance.

Communications byte streams have been extensively optimized, particularly at the SamC level. This interface provides better performance than any but the most carefully crafted user-written routines.

Who Cannot Use Communications Byte Streams

Applications which use a *synchronous* RS-232-C communication protocol cannot use communications byte streams. If you are writing a synchronous application, skip the next two main sections, and read "Writing Synchronous Communications Applications," later in this chapter.

Using Communications Byte Streams at the Device-Independent Interface Level

The Sequential Access Method (SAM)

This section describes what you need to know to access communication ports from your application at the level of OpenByteStream, ReadBsRecord, WriteBsRecord, and the other device-independent SAM procedures.

Device Specifications

To support port expander modules and multi-processor servers, SAM recognizes more devices than are found on a typical workstation.

SAM accepts the device specifications described in Table 18-1.

Table 18-1. SAM Communications Device Specifications

Channel Synonyms			Processor Channel	Device
A	0	0A	A	Workstations, SRP/XE
B	1	0B	B	Workstations, SRP/XE
C	2		C	SRP/XE
D	3		D	SRP/XE
E	4		E	SRP/XE
F	5		F	SRP/XE
G	6		G	SRP/XE
H	7		H	SRP/XE
I	8		I	SRP/XE
J	9		J	SRP/XE
	1A		n/a	Port Expander 1, Channel 1
	1B		n/a	Port Expander 1, Channel 2
	1C		n/a	Port Expander 1, Channel 3
	1D		n/a	Port Expander 1, Channel 4
	2A		n/a	Port Expander 2, Channel 1
	2B		n/a	Port Expander 2, Channel 2
	2C		n/a	Port Expander 2, Channel 3
	2D		n/a	Port Expander 2, Channel 4

These specifications are valid in SAM device specifications for *[comm]* or *[ptr]* byte streams, with or without an associated configuration file name. For example, the following are all valid:

```
[comm]1A
[ptr]2b&<mydir>myconfig.sys
[comm]1D&cfile.cnfg
```

In the first example above, no configuration file is specified, and communications byte streams looks for a default configuration file named as follows:

```
[Sys]<Sys>Comm1AConfig.Sys
```

Other specifications may be valid in the future, as new hardware is introduced. Programs should treat device specifications as uninterpreted strings, allowing the system to accept or reject them.

Configuration Files

To use communications byte streams at the device-independent level, it is necessary to have a configuration file for each channel. More than one channel can use the same configuration file, in which case that file's name must be specified explicitly on the OpenByteStream request for each channel. Use the **Create Configuration File** command to create or edit these configuration files. (See your operating system documentation for details.)

Some channels on port expander modules support NRZI encoding. Workstation processor channels do not. See your port expander hardware documentation for more information. If a program attempts to open a channel on which a specified option is unsupported, status code 7 ("Not implemented") is returned.

Notes on Customizing SAM for Communications

The Sequential Access Method (SAM) is a subroutine package that can be configured to include or exclude support for particular devices. Each type of device has a corresponding type of byte stream. You can produce a custom configuration of SAM by editing and reassembling the file [Sys]<Sys>SamGenAll.asm.

The default configuration of SAM does not include communications byte streams. Therefore, if you link your application with CTOS.lib without providing an alternative SamGen module, your application cannot use the communications-specific procedures in SamC.

If you explicitly include [Sys]<Sys>SamGenAll.obj in your link, that file is used in place of the version of SamGen in CTOS.lib. This configuration includes all types of byte streams (except *[tape]* byte streams), including *[comm]* and *[ptr]* byte streams).

You can choose your own subset of the byte stream types you need by editing and reassembling a SamGen source file, then including the resulting

object file in your link. The procedure for doing this is described in Chapter 17, "Building a Customized SAM" in this manual.

Using Communications Byte Streams at the Device-Dependent Interface Level

The Communications Sequential Access Method (SamC)

Communications byte streams itself is referred to as *SamC*, to distinguish it from the rest of the Sequential Access Method (SAM).

Programs that are distinctly communications-oriented can take advantage of a more powerful and flexible set of services than those available at the device-independent level. Programs which are not communications-oriented (such as the CTOS Executive) can still use communications byte streams via SAM, just like any other type of byte stream.

In addition to playing its role as part of SAM, SamC supports additional calls that are not appropriate for other types of byte streams. Most of these additional calls are described in this chapter. All the communications byte stream interfaces, including those called by SAM, are described in the *CTOS/Open Application Programming Interface Specification*.

The interfaces provide control over communications byte streams at a lower level (closer to the hardware) than do the SAM device-independent interfaces. Although more complex to use, they constitute a more complete set of services. Applications may also supplement the SAM interfaces by occasionally using SamC interfaces.

SamC can act as a complete replacement for SAM if only communications byte streams need to be supported. (How to do this is discussed below under "Using SamC Without SamGen.") Used in this fashion, communications byte streams is a general-purpose device driver for asynchronous RS-232-C communications. It can form the heart of virtually any communications product except those that use synchronous communications protocols. Both half and full duplex communications are supported efficiently with a variety of line control and data editing options. Among other conveniences, using communications byte streams frees you from writing interrupt service routines.

Using SamC Without SamGen

Programs that use SamC directly can avoid including any modules from the device-independent parts of SAM. If none of the device-independent SAM interfaces are used, several modules that support these interfaces are excluded at link time, saving memory. In addition, this removes the need to configure SAM using SamGen.asm.

To avoid including other parts of SAM besides SamC itself, use *only* the SamC interfaces. These interfaces are listed under the heading "Communications Programming" in Figure 2-1 in the *CTOS/Open Application Programming Interface Specification*. Do not use the device-independent SAM interfaces listed under "Sequential Access Method" in the same figure.

In addition, to avoid including SamGen from CTOS.lib automatically, you must include in your program the following global public data declarations:

```
Byte  tagCommWrite = 1;
Byte  tagCommRead = 2;
Byte  tagCommModify = 3;
Byte  tagPtrWrite = 4;
```

Otherwise, the Linker includes parts of SAM even though you do not call them.

Some Special Features of SamC

This section describes some of the features of SamC, which distinguish it from the more generic SAM interface.

Asynchronous Interface

The FillBufferAsyncC, FlushBufferAsyncC, and CheckPointBsAsyncC procedures are variants of FillBufferC, FlushBufferC, and CheckPointBsC, respectively. Because communications byte streams is a subroutine package, you cannot issue asynchronous (nonblocking) requests to it as you can with disk or keyboard byte streams, for example. This ability is frequently useful, and sometimes required.

The asynchronous SamC interface allows an application to check for the availability of data, without having to wait if none is available.

Additional parameters allow the caller to specify what SamC should do if the operation cannot be completed immediately. See the *CTOS/Open Application Programming Interface* for more information on the individual calls.

AcquireByteStreamC Interface (Low-Level Open)

The OpenByteStream and OpenByteStreamC interfaces require a configuration file which contains the communications line configuration parameters (baud rate and so on) shown in Table 18-2. AcquireByteStreamC is a lower-level interface that accepts an in-memory structure corresponding to the contents of the configuration file. This allows applications to avoid using configuration files if they do not truly need them.

AcquireByteStreamC also provides more control over the buffer sizes chosen for the receive and transmit queues. Under OpenByteStreamC, the user supplies a single memory area of a chosen size, which OpenByteStreamC divides up between receive and transmit queues. AcquireByteStreamC allows the program to specify each queue explicitly.

Table 18-2. Communications Configuration Descriptor

(Page 1 of 2)

Offset	Field	Size (bytes)	Description
0	type	1	The types and their values are Parallel Printer 1 Serial Printer 2 Communications 3
1	baudRate	2	The transmit baud rate (0-19200 bits per second)
3	stopBits	2	The stopBits value is 0-2.
5	charSize	1	The charSize value is 5-8.
6	parity	1	No parity 0 Even parity 1 Odd parity 2 One parity 3 Zero parity 4
7	lineControl	1	The line control modes and their values are No line control 0 XON/XOFF mode 1 Clear to Send (CTS) mode 2 Both XON/XOFF and CTS modes 3
8		2	Unused
10	TxMap	1	The output new line mapping modes and their values are Binary mapping 0 New line 1 Carriage Return 1 Carriage Return/Linefeed 2

Table 18-2. Communications Configuration Descriptor

(Page 2 of 2)

Offset	Field	Size (bytes)	Description
11	RxMap	1	The input new line mapping modes and their values are
			Binary 0
			NewLine 1
			Carriage Return 1
			Carriage Return/ Linefeed 2
12	TxTimeOut	2	Time in seconds before a timeout on a transmission
14	RxTimeOut	2	Time in seconds if no characters are received before a status code indicating "Device not ready" is returned
16	fEOF	1	TRUE indicates that bEOF is active
17	bEOF	1	A character signifying the end of file
18	sTab	1	The size (number of spaces) of the expanded tab
19	cCharsPerLine	1	The new line mode is generated (based on TxMap) after this many bytes
20	sbTxItnFile (79)	1	The character translation file name
99	fNRZI	1	TRUE indicates Non Return to Zero Inverted mode (NRZI). Not supported on workstation port A or B.
100	RxBaudRate	2	The receive baud rate (0-19200 bits per second)

Listing 18-1 shows a code fragment that opens a serial communications channel using AcquireByteStreamC.

```
/*
Set up the Communication Configuration Descriptor for
port A
*/
CommCfgBlk.Type = 3;           /* Type = Communications */
CommCfgBlk.BaudRate = 9600;    /* Transmit at 9600 baud */
CommCfgBlk.StopBits = 1;       /* 1 stop bit */
CommCfgBlk.CharSize = 8;       /* 8 bits per character */
CommCfgBlk.Parity = 0;         /* Parity = None */
CommCfgBlk.LineControl = 1;    /* XON/XOFF */
CommCfgBlk.TxMap = 2;          /* CR/LF */
CommCfgBlk.RxMap = 1;          /* Newline */
CommCfgBlk.TxTimeOut = 5;     /* Tx timeout = 5 seconds */
CommCfgBlk.RxTimeOut = 5;     /* Rx timeout = 5 seconds */
CommCfgBlk.fEOF = TRUE;        /* bEOF is active */
CommCfgBlk.bEOF = 4;           /* EOF Character */
CommCfgBlk.sTab = 8;           /* Tab expansion */
CommCfgBlk.cCharsPerLine = 132; /* 132 chars per line */
CommCfgBlk.sbTxltnFile[0] = 0;  /* No translation file */
CommCfgBlk.fNRZI = FALSE;      /* Not NRZI */
CommCfgBlk.RxBaudRate = 9600; /* Receive at 9600 baud */

/*
 Allocate memory for the Port A BSWA and Queues
*/
selectorof(pBSCommA) = snMemArea;
CheckErc(ExpandAreaSL(130, snMemArea,
                      &(offsetof(pBSCommA)) ));
selectorof(pbTQA) = snMemArea;
CheckErc (ExpandAreaSL (1024, snMemArea,
                        &(offsetof(pbTQA)) ));
selectorof(pbRQA) = snMemArea;
CheckErc (ExpandAreaSL (1024, snMemArea,
                        &(offsetof(pbRQA)) ));
cbTQ = (1024 - 12) / 2; /* wordsize with 12 rsvd bytes */
cbRQ = (1024 - 12) / 2;

/*
 Open first port using AcquireByteStreamC
*/
CheckErc (AcquireByteStreamC (pBSCommA, rgbPortID, 7,
                 modeModify, &CommCfgBlk, pbTQA, cbTQ,
                 pbRQA, cbRQ, 0, 0));
```

Listing 18–1. Opening a Channel with AcquireByteStreamC

Ability to Change Parameters Dynamically

The ReadByteStreamParameterC and WriteByteStreamParameterC operations allow a program to query and change the values of various communications line parameters dynamically.

Ability to Query and Set Status Lines

The RS-232-C standard defines some additional status lines which may be needed when you are dealing with modems or special hardware. Communications byte streams provides an interface to access or change the state of these lines, using the ReadStatusC and WriteStatusC operations.

Writing Synchronous Communications Applications

Access Below the Byte Stream Level (CommLine)

Communications byte streams does not support the use of the serial controller in synchronous mode. To write an application that uses a synchronous communication protocol, it is necessary to interface directly with the operating system at a level below communications byte streams.

The following operations are part of the operating system's support for serial ports. (Communications byte streams itself uses these requests.)

InitCommLine
ResetCommLine
ChangeCommLineBaudRate
TerminateCommLine
ReadCommLineStatus
WriteCommLineStatus

These operations address three objectives:

- workstation-independent applications that do not have to be relinked for each new machine
- protected-mode-compatible raw interrupt routines
- XE/SRP compatibility

The design of the CommLine-level interface places all workstation-specific code inside the operating system Comm Nub. This allows applications to use the interface without regard for the type of hardware on which they run.

Guidelines for Compatible Synchronous Applications

The details of the CommLine interface and interrupt service routine calling conventions are explained later in this chapter. The following checklist enumerates the most important requirements for portability. Synchronous communications applications should have:

- No code that parses or validates the device specification string before it is passed to InitCommLine. Let InitCommLine validate the string, so that the program will continue to work if the list of valid device specifications recognized by the operating system is expanded in the future.
- No knowledge of which serial port it is actually using. The purpose of InitCommLine is to allow the application to work identically on any port, on any type of workstation.
- No code that uses any input/output port addresses except the two port addresses returned by InitCommLine. Using other port addresses implies a machine dependency.
- No calls to GetpStructure or GetModuleId. It is not necessary to know details of the hardware configuration when using InitCommLine.
- No calls to MediateIntHandler, which should never be used in a communications interrupt service routine.
- Calls to PSend only in mediated communications ISRs, not in raw communications ISRs.

Applications that abide by these limits on their knowledge should be portable.

The CommLine Interface

The following sections describe each of the procedures needed to open a communications channel using the CommLine interface.

The procedural interfaces for the InitCommLine, ResetCommLine, and ChangeCommLineBaudRate requests are described in detail in the *CTOS/Open Application Programming Interface Specification*. Their use is described next.

InitCommLine

InitCommLine assigns the caller to a physical channel on a serial controller by parsing the device specification passed to it. You should treat this specification as an uninterpreted string, so that your application will continue to work when new hardware modules (with new forms of file specifications) are introduced.

On success, InitCommLine returns a handle for the open communications line. It also returns two port addresses, a control port and a data port, for the channel. These addresses can be used to initialize the serial controller.

In some CTOS-based operating systems, InitCommLine also returns additional information, such as the type of serial controller installed. This additional information should be used with care, because it is only available on certain operating system versions. See your operating system documentation for more information.

Note that InitCommLine does not tell the caller which channel of the serial controller it uses. The application does not need to know. The InitCommLine interface performs all general serial controller operations that could require knowledge about serial controller channels. For example, the Comm Nub resets the serial controller after each interrupt.

The application must still perform some operations directly on the serial controller, using the two returned port addresses. InitCommLine does not fully initialize the channel (although it does reset it), since it does not know all the parameters the application needs.

Note that the only parameters provided to InitCommLine are those dealing with external hardware (outside the serial controller). This hardware (baud rate timers and external control registers) is InitCommLine's

responsibility because it varies from machine to machine. The serial controller, however, is invariant: all CTOS machines use the same Intel 8274 (or software-equivalent) serial controllers.

In addition, the serial controllers on some CTOS machines (or option modules) provide a superset of the standard Intel 8274 serial controller interface. Care should be taken to ensure that such a controller is present before using their additional features.

ResetCommLine

ResetCommLine closes a communications channel opened by InitCommLine. The only parameter passed to ResetCommLine is a handle returned by InitCommLine.

You cannot issue ResetCommLine until you have successfully completed an InitCommLine operation for that channel. InitCommLine acquires the channel for you (and resets it so you have a chance to initialize it to your specifications before you start taking interrupts). ResetCommLine gives the channel back to the operating system, making it available for other users.

All channels opened by InitCommLine should be closed by ResetCommLine before the program exits.

ChangeCommLineBaudRate

ChangeCommLineBaudRate is used to change InitCommLine's baud rate parameters dynamically. ChangeCommLineBaudRate should be used only on channels that were opened using InitCommLine. SamC clients should use WriteByteStreamParameterC, instead.

The serial controller is not affected by ChangeCommLineBaudRate.

CommLine Status Procedures

The procedures in the following sections are system-common procedures, not requests. Therefore, they can be called from an interrupt service routine as well as from the main program. The procedural interfaces for these operations are described in detail in the *CTOS/Open Application Programming Interface Specification*.

These procedures should be used only on channels that were opened using InitCommLine. SamC clients should use WriteByteStreamParameterC, instead.

ReadCommLineStatus: This procedure allows certain RS-232-C signals, whose function is not defined by the serial controller, to be queried by the application in machine-independent fashion.

WriteCommLineStatus: This procedure allows certain RS-232-C signals, whose function is not defined by the serial controller, to be raised or lowered by the application in machine-independent fashion.

Opening a Serial Communications Channel with InitCommLine

There are three steps to opening a communications channel with InitCommLine:

1. Building the Communications Line Control Block.
2. Calling InitCommLine and verifying that it was successful.
3. Performing any additional serial controller initialization required.

The following sections describe each of those functions.

Building a Communications Line Control Block

The Communications Line Control Block structure contains the information required by InitCommLine to initialize the communications channel. The format of the Communications Line Control Block is shown as part of the description of InitCommLine in the *CTOS/Open Application Programming Interface Specification*.

The following sample procedure builds a Communications Line Control Block. The variable type FlcbType is a Frame Level Control Block structure defined by the program. It contains various items of information about the communications channel. For a more complete listing of the program from which this example (and the following two) are drawn, see Appendix D.

```
void  PrepCommBlk(FlcbType *pflcb, char iSizeRetBlk)
{
  cbcommlcb = sClcbTypeMax;
  if (iSizeRetBlk == BTOS)
     cbcommlret = sInitCommLineRetTypeMax; /* 23 bytes */
  else if(iSizeRetBlk == BTOSII2)
     cbcommlret = sBTOSII2RetTypeMax;      /* 19 bytes */
  else
     cbcommlret = sInitCommLineRetType;    /* 6 bytes */

   /* set some not-always-supported fields to FALSE */
   commlret.dmahardware = 0;
   commlret.chiptype = 0;
   /* set up the config. block */
   commlcb.pdsbx   = (char _far *)pflcb;

   commlcb.fdma = false;
   commlcb.ptxisr  = (void _far *)pIsrTxData;
   commlcb.prxisr  = (void _far *)pIsrRxData;
   commlcb.psprxisr =(void _far *) pIsrRxSpec;
   commlcb.pextisr = (void _far *)pIsrExStat;

   commlcb.frawtx   = true;
   commlcb.frawext  = true;
   commlcb.frawrx   = true;
   commlcb.frawsprx = true;
   commlcb.baudtx  = iBaudRate;  /* 0 for ext clocks */
   commlcb.baudrx  = iBaudRate;  /* 0 for ext clocks */
   commlcb.fnrzi    = false;

   commlcb.fx2l = false;
   commlcb.ftdi = false;
   commlcb.ftdixlat= false;
}
```

Listing 18–2. Building a Communications Line Control Block

Initializing the Communications Line

All CTOS-based operating systems return the same information in the first six bytes of the InitCommLine return information. These six bytes are defined as follows:

Offset	Description
0	CommLine Handle (Word)
2	Control Port Address (Word)
4	Data Port Address (Word)

Because different CTOS-based operating systems return different amounts of information after the first six bytes, a portable program should allocate space for the largest possible returned block (currently 23 bytes) even if it only makes use of the six standard bytes.

The example calls InitCommLine with each possible returned size in turn, from largest to smallest. This allows the program to make use of whatever extra information may be available.

```
char  InitComm(FlcbType *pFlcb)
{

Word   commerc ;
char       iSizeRetBlk;

fdoinitcomm = true;
iSizeRetBlk = BTOSII2;

while   (fdoinitcomm) {     /* InitCommLoop */
   if (cbcommspec == sFourPortChannel)
      iSizeRetBlk = BTOS;
   PrepCommBlk (pFlcb, iSizeRetBlk);

   commerc=InitCommLine(&commspec[0], cbcommspec,
            &commlcb, cbcommlcb, &commlret,cbcommlret);
   if (commerc == ercRetArea2Large)  {
      iSizeRetBlk = BTOS;
      PrepCommBlk (pFlcb, iSizeRetBlk);
      commerc = InitCommLine(&commspec[0], cbcommspec,
               &commlcb, cbcommlcb, &commlret,cbcommlret);
      }
   if (commerc == ercRetArea2Large)  {
      iSizeRetBlk = CTOS;
      PrepCommBlk (pFlcb, iSizeRetBlk);
      commerc = InitCommLine(&commspec[0], cbcommspec,
                             &commlcb, cbcommlcb,
                             &commlret,cbcommlret);
      }
```

continued...

```
    CheckErc(commerc);
    fdoinitcomm = false;
    }
return (true);
}
```

Listing 18–3. Opening a Channel with InitCommLine

Initializing the Serial Controller

In addition to calling InitCommLine, the program must perform a certain amount of serial controller initialization. The example below performs some basic initialization functions both for the normal 8274-type controllers and for the 82530 controllers used in some port expansion modules.

The procedure uses a conditional flag in the extended InitCommLine return information to determine whether an 82530 is present. The program should be sure to initialize this flag to zero, so that it only becomes non-zero if the operating system explicitly makes it so.

For information on serial controller registers and their use, refer to the technical reference documentation for your workstation, or to the technical reference for the controller itself.

```
void   InitFdxIo(FlcbType *pFlcb)
{
 Word    port;
 char        i;

i = InitComm (pFlcb);

pFlcb->iocommctl = commlret.iocommctl;
pFlcb->iocommdata = commlret.iocommdata;
wcommlinehandle = commlret.commlinehandle;
wchiptype = commlret.chiptype;

port = pFlcb->iocommctl;
pFlcb->fxmitcomplete = false;
frxspec = false;
frcvmsg = false;
```

continued...

```
rgpflcb = pFlcb;

/* turn off interrupts */
CLI;
/* set chip to bit-sync mode */
LockOut(port,commWr4);
LockOut(port,comm4Xl+comm4BitSync);
/* set end of frame flag */
LockOut(port,commWr6);
LockOut(port,zFlag);
LockOut(port,commWr7);
LockOut(port,zFlag);
/* enable interrupts.  Receive int on first char or
   on special condition
*/
LockOut(port,commWrl);
LockOut(port,commlRxEnable+commlTxIntEnb + commlExIntEnb
        + commlStatVect);
/* 8-bit chars, other receive params */
LockOut(port,commWr3);
LockOut(port,comm3Rx8 + comm3AutEnb + comm3RxEnb +
        comm3RxCrcEnb + comm3Hunt);
/* enable int on next receive, set next write to reg 5 */
LockOut(port,commRxint);
LockOut(port,commRxint+commWr5);
/* set transmit params */
LockOut(port,comm5Dtr + comm5Tx8 + comm5CrcCcitt +
        comm5TxEnb + comm5TxCrcEnb);
/* error reset and set next write to reg 4 */
LockOut(port,commErrRes+commWr4);
LockOut(port,comm4Xl+comm4BitSync);

if (wchiptype == 1) {       /* 82530 only  */
   LockOut(port,commWrl0);
   LockOut(port,comml0CrcPreset); /* not NRZI */
   LockOut(port,commWrl5);
   LockOut(port,comml5AbrtIE + comml5TxUndIE +
           comml5CtsIE + comml5DcdIE + comml5SyncHuntIE);
   LockOut(port,commWrll);
   LockOut(port,commllRxRTxC+commllTxTRxC);
   }
/* restore interrupts */
STI;
}
```

Listing 18–4. Initializing the Serial Controller for Synchronous Communication

CommLine Interrupt Service Routines

This section describes the interrupt handlers needed for successful CommLine-level communications.

The part of the operating system that dispatches communications interrupts is called the Comm Nub. The Comm Nub receives control directly from the serial communications hardware interrupt. The Comm Nub then transfers control to your interrupt service routine, according to the instructions your program gave the operating system when it performed the InitCommLine operation. When your ISR has completed processing the interrupt, it returns to the Comm Nub.

Communications ISRs differ somewhat from other types of ISRs, because the Comm Nub performs some extra services for them. This section describes these special communications-oriented ISRs.

The Four Interrupt Service Routines for a Communications Channel

Each channel that uses the CommLine interface must have four interrupt service routines defined for it. They are:

1. A transmit ISR, to transmit characters and manage the transmit buffer.

2. A receive ISR, to receive characters.

3. A "receive special" ISR to handle receive errors and end-of-frame conditions.

4. An external status ISR to handle RS-232 signal transitions and error conditions.

Listing 18-5 shows an example of a raw receive ISR. Examples of the other three ISRs for a communications channel can be found in Appendix D.

Note that the example uses a macro, ZIN, instead of the normal processor IN instruction. This macro flushes the processor's prefetch queue, then executes the IN instruction. Its counterpart, ZOUT, performs the same function for the OUT instruction. Use of these macros ensures that all previous INs and OUTs have been completed before beginning a new one. The macro definitions for ZIN and ZOUT can be found in Appendix D.

```
;-------------------------------------------------------
;   Receive data
;     routine to handle receive char available interrupt.
;   Input:
;     [DS:BX] - point to FLCB
;   Output:
;     None
;-------------------------------------------------------

IsrRxData             PROC        FAR
 MOV  DX,WORD PTR[BX+flcbioCommData]
 %ZIN  (AL,DX)                          ; receive a char
 MOV  CX,WORD PTR[BX+flcbiRcvData] ;check for full buffer
 CMP  CX,WORD PTR[BX+flcbcbRcvMax]
 JGE  IsrRxData_x                       ;if not full
 TEST fRCVmsg, 0ffh                     ;check for end of text
 JNZ  IsrRxData_y
 LES  SI,DWORD PTR[BX+flcbpRcvData] ;put data in buffer
 ADD  SI,CX
 MOV  ES:BYTE PTR[SI],AL

IsrRxData_x:
 INC  WORD PTR[BX+flcbiRcvData]     ;increment buffer pos.

IsrRxData_y:
 JMP  IsrTxData_0  ;checks for empty send buffer, returns
                   ;if buffer is not empty

IsrRxData             ENDP
```

Listing 18–5. A Receive Interrupt Service Routine

Raw Communications Interrupt Service Routines

Raw communications interrupt service routines can be written only in assembly language. Partly because they are in assembly language, and partly because the operating system does less work dispatching them, they are much more efficient than mediated interrupt service routines. If it is worth the extra work required to use the CommLine interface, it is almost always worth the extra work to write raw communications ISRs, as well.

In real mode, a raw interrupt handler uses the stack of whatever process happened to be running when the interrupt occurred. For this reason, raw interrupts must carefully control their stack depth. A raw interrupt that

uses too much stack space can overwrite the memory of any process in the machine, including the operating system processes. This behavior can be very hard to trace to its source, because it is so hard to reproduce exactly.

In protected mode, raw interrupt handlers use one of the operating system's stacks. Therefore, the stack depth restriction is somewhat relaxed, but raw ISRs should still keep their stack usage to a minimum.

Conditions on Entry. The new Comm Nub saves all the registers before it passes control to the ISR. In addition, the Comm Nub performs two other services for the ISR. It loads DS:BX with the pDsBx value that was specified in the InitCommLine request and it puts the port address of the channel's control port in DX.

Interrupts must remain disabled throughout a raw interrupt handler's execution.

Exiting From the Raw ISR. When it transfers control to the ISR, the nub performs a CALL to the raw ISR. Therefore, the ISR must always exit with a RET to the nub. The nub then restores all the interrupted program's registers.

All end-of-interrupt functions are handled by the Comm Nub. The nub always sends EOI to the serial controller and to the interrupt controller(s).

In addition, the ISR can request the nub to perform a PSend operation for it by leaving values in certain registers when it exits.

If AX is non-zero when the ISR exits, the Comm Nub performs a MediateIntHandler call followed by a PSend operation on behalf of the ISR. For this to occur, the ISR should leave the following values in the registers listed below:

AX	destination exchange for the PSend operation
DS:BX	value of the pMsg parameter of the PSend operation (the address of the message)

The ISR should never call MediateIntHandler on its own.

Mediated interrupt handlers can be written in higher-level compiled languages. In general, however, communications ISRs should be raw. If the extra performance is worth the trouble of using the CommLine interface, the extra performance gained by using raw ISRs is equally worth the trouble.

Conditions on Entry. A mediated interrupt procedure takes a single pointer argument, as described in its function declaration:

```
void fooISR(Pointer pDsBx);
```

The pDsBx pointer is the one supplied to InitCommLine. This value is also placed in DS:BX, as with raw interrupt handlers. In most high-level languages the contents of BX are irrelevant, but in medium model the contents of DS can be vital.

Typically, pDsBx is the address of some global variable in the application's DGroup. However, there is no requirement that this be the case unless the ISR needs to reference the application's data segment. If a medium-model mediated interrupt handler references any static variables at all, pDsBx must be a pointer to something in DGroup so that DS will be set up correctly.

Exiting From the Mediated ISR. As with raw interrupts, the Comm Nub performs all EOI processing.

In a mediated ISR you can perform your own PSend operation(s) to wake up processes. Unlike the case with raw ISRs, there is no way to ask the Comm Nub to do this for you upon return from a mediated interrupt.

Inside CTOS: A Guide to Performance Enhancement

This section describes what the operating system has to do to support your interrupt service routines. Familiarity with what is actually going on inside the operating system can help you write more efficient code.

Interrupt Dispatching (The Comm Nub)

The Comm Nub is the operating system's communications interrupt dispatcher. Because a single hardware interrupt vector supports multiple channels, which may belong to different applications, the operating system must be involved in vectoring each hardware interrupt to its proper interrupt service routine.

Raw Interrupts

The most direct form of vectoring takes place with raw interrupts. With raw interrupts, the Comm Nub acts simply as a short prologue and epilogue to the interrupt service routine itself.

The Comm Nub saves the registers, reads the serial controller's status to determine which channel the interrupt is for and what type of interrupt it is, and calls the appropriate raw interrupt service routine.

Arguments to the interrupt service routine, and results from it, are passed in registers.

Upon return from the interrupt service routine, the Comm Nub can wake up a process (using PSend) for the interrupt service routine if it is requested to do so. If this is not required, the Comm Nub simply performs EOI for the serial controller and the interrupt controller(s), then restores the interrupted process's registers and does an IRET to the interrupted process.

To write efficient raw interrupt service routines, observe the following guidelines:

- The overhead of the PSend, which involves a process scheduling and context switch operation, usually exceeds the overhead of the rest of the Comm Nub and of the interrupt service routine itself. For this reason, an interrupt service routine should awaken a process only when necessary, not on every interrupt.

  Thus the interrupt service routine should communicate with the process associated with it only as much as is absolutely necessary. For example, the receive interrupt service routine usually has a multi-character buffer that it fills before waking the process, and the transmit interrupt service routine has a buffer that it empties.

A typical mistake is for a transmit ISR to wake the transmitting application process (which is waiting for buffer space to become available) as soon as one byte of space is available in the transmit buffer. When the buffer is full, this results in a context switch after each character is sent. A better scheme is to wait until the buffer is a third or half empty, and refilling the buffer with characters at that time.

- Code the interrupt service routines as tightly as possible. One of the ISRs runs every time a character is sent or received, and a few instructions can make a visible difference in performance at a high baud rate or when multiple channels are in use simultaneously. Let the Comm Nub set up DS and BX for you so that the ISR can quickly locate the data structure that it needs when it services the interrupt.

- Code the entire ISR in assembly language. Calling procedures written in higher-level languages from inside a raw ISR may defeat the benefit of writing a raw ISR in the first place.

Mediated Interrupts

Mediated interrupts suffer the overhead of raw interrupts plus additional costs, because a mediated interrupt begins its life as a raw interrupt. Instead of calling a raw interrupt service routine, the Comm Nub calls the Mediated Interrupt Nub, which switches control to the operating system's interrupt stack. Then the user's mediated interrupt service routine is called. Thus, a mediated ISR always runs on the operating system's interrupt stack, in both real and protected mode.

The costs of writing an interrupt service routine in a higher-level language should also be considered. Assembly language is usually more efficient.

In addition, mediated interrupt handlers can themselves be interrupted, because they run with interrupts enabled (unless the ISR explicitly disables them). This fact can have serious consequences in heavily-loaded systems. Note however, that another interrupt of the same type (for example, another communications interrupt) cannot occur until the first interrupt is completely finished, because end-of-interrupt (EOI) acknowledgment is not sent to the serial controller and the interrupt controller(s) until the ISR returns to the Comm Nub.

The advantage of having interrupts enabled during mediated ISRs is that an interrupt of a different type, having a higher hardware priority, can occur during servicing of a lower-priority interrupt. In this case, the fixed priority associated with the attachment of peripherals to interrupt levels on the interrupt controller determines which ISRs can interrupt other ISRs; CTOS process priority plays no role. This arrangement permits peripherals with less patience to be serviced during execution of a less critical ISR and makes it less important to keep such ISRs brief. (*Patience* is the amount of time that can safely elapse before a interrupt is serviced by its ISR.) A communications ISR, for example, can run during execution of a keyboard or disk ISR. This use of priorities may be especially important if the disk ISR takes a comparatively long time to complete.

This benefit is not a good argument for making communications ISRs mediated, however. Communications ISRs must be brief in order to sustain high baud rates. Mediated interrupts are more appropriate for devices that interrupt less frequently, such as DMA devices or the keyboard.

19
Interrupt Handlers

This chapter describes the types of interrupt handlers available under CTOS, and gives some guidelines for writing each type.

Interrupts and Exceptions

There are several related mechanisms which can interrupt CTOS program execution. These mechanisms fall into two broad groups: interrupts, and exceptions.

An *interrupt* is an asynchronous event, typically triggered by an external device which needs attention.

An *exception* is the CPU's response when it detects certain conditions in the course of executing an individual instruction.

Both interrupts and exceptions cause the processor to suspend the currently-executing program. The processor then transfers control to a special routine, written to handle that particular interrupt or exception.

The main difference between an interrupt and an exception is its origin. An exception is (at least indirectly) software-generated, and can be reproduced by re-running the program that caused it. An interrupt, however, is hardware-generated and therefore independent of any particular program.

Types of Interrupts

True interrupts come in two flavors: maskable and nonmaskable. Both are caused by external hardware.

Maskable interrupts are the most common type of interrupt, and are generated by the interrupt controller hardware in response to a request from a peripheral device. Maskable interrupts can be suppressed temporarily (masked) by software, hence the name.

It is this type of interrupt that most hardware device interrupt handlers service. For example, when a program installs an interrupt handler to process interrupts from a line printer, it services a maskable interrupt.

Nonmaskable interrupts are also generated by the interrupt controller hardware, but in response to a disastrous hardware event, such as a memory parity fault. This type of interrupt is handled by the operating system (generally by crashing the system), and is almost never serviced by an installed interrupt handler.

Types of Exceptions

Exceptions come in several flavors. The ones that could be useful to an application programmer are programmed exceptions, faults and traps. All three are caused by software.

A *programmed exception* is the result of an explicit software action. For example, the INT instruction causes a programmed exception. This type of exception is also sometimes called a *pseudo-interrupt* or a *software interrupt*. These events are actually treated as exceptions by the processor, though, not as true interrupts. For consistency with other CTOS documentation, programmed exceptions are called pseudo-interrupts elsewhere in this manual.

A *fault* is an exception that is reported before the instruction which caused the exception. When the processor reports a fault, it has detected a problem either before or during execution of the offending instruction. The processor restores itself to the state it was in before the instruction, so that the instruction can be restarted if desired. An example of a fault is the General Protection fault, wherein the processor detects that a program is trying to load an illegal value into a segment register.

A *trap* is an exception that is reported immediately after the instruction which caused the exception. Because it is reported after the instruction executed, it is not possible for the processor to restore itself to a previous state. An example of a trap is the Overflow exception, which can be

caused when the product of an addition instruction overflows the register that should contain it.

When to Use an Interrupt or Exception Handler

Interrupt handlers and exception handlers should be used frugally. There are few circumstances that truly require using one, and their use should be limited to those circumstances. Some of the uses for interrupt and exception handlers are:

- Low-level communication with peripheral devices
- Special fault-handling routines in application software
- System-wide fault or pseudo-interrupt handling routines.

Types of Interrupt and Exception Handlers

Not only are there multiple types of interrupts and exceptions, there are multiple types of interrupt handlers to process them. This section maps the types of interrupt handler to the appropriate interrupt and exceptions for them to handle.

In protected mode, the essential dividing line is drawn between interrupts and exceptions. In CTOS, interrupt handlers which service true hardware interrupts are called *interrupt handlers*. Those which service exceptions are called *trap handlers* or *pseudo-interrupt handlers*.

Types of Interrupt Handlers

Two primary types of hardware interrupt handlers can be installed: raw and mediated. Both are installed by a call to the SetIntHandler procedure. The program specifies which type of interrupt handler it is installing when it calls SetIntHandler.

Any interrupt handler which is installed using SetIntHandler is global in scope. That is, it services all occurrences of its interrupt throughout the system, no matter what their source.

Only those interrupts which were defined to CTOS at operating system generation (SYSGEN) can be serviced by interrupt handlers. If you want to establish a routine to handle an interrupt that was previously undefined to CTOS, you must make it an exception handler. See the description of "low memory allocation" in your operating system documentation to determine which interrupts are defined in your system.

Raw Interrupt Handlers

A *raw* interrupt handler is a traditional-style interrupt handler, and should be written in assembly language. Interrupts are disabled when a raw interrupt handler receives control, and should remain disabled throughout the interrupt handler's execution.

The only registers that are defined on entry to a raw interrupt handler are CS, IP and DS. In real mode, all other registers must be saved before they are used. In protected mode, the registers need not be saved, because the processor saves them when it switches context to the interrupt handler.

A raw interrupt handler must also perform any needed end-of-interrupt processing, such as resetting the programmable interrupt controller and the device that requested service. Also, a raw interrupt handler must return using an interrupt-return (IRET) instruction, instead of procedural-return (RET) instruction.

A raw interrupt handler can make only one operating system call, MediateIntHandler. This call converts the interrupt handler to the mediated type. After a raw interrupt handler calls MediateIntHandler, it must follow the guidelines for mediated interrupt handlers.

Mediated Interrupt Handlers

A *mediated* interrupt handler is, as its name implies, mediated by the operating system. When a mediated interrupt handler receives control, interrupts are enabled, and the contents of all registers have been saved by the operating system.

Mediated interrupt handlers can be written in a high-level language and can make operating system calls. They should still be kept as brief as possible, though. Also, because interrupts are enabled, mediated

interrupt handlers can be interrupted. This fact should be kept in mind when writing them.

On exit from a mediated interrupt handler, the operating system restores the contents of all registers and sends an end-of-interrupt signal (EOI) to the programmable interrupt controller. However, if the interrupt handler controls a device, it must send any required end-of interrupt signal to that device before exiting.

A mediated interrupt handler returns using a procedural-return (RET) instruction. This allows a mediated interrupt handler to be a standard high-level language procedure.

Communications Interrupt Handlers

The interrupt handlers used for synchronous communications are a special case, and are not covered in this chapter. Communications interrupt handlers are special because all serial port interrupts are first routed through the operating system's Comm Nub. The Comm Nub performs some setup before transferring control to the interrupt handler, then performs some cleanup and EOI processing after the interrupt handler returns.

For detailed information on communications interrupt handlers, see Chapter 18, "Communications Programming."

Types of Exception Handlers

An exception handler can be considered either a trap handler or a pseudo-interrupt handler. Whether a given piece of code is considered a trap handler or a pseudo-interrupt handler depends mainly on the event that causes it to execute.

Trap Handlers and Pseudo-Interrupt Handlers

If a program deliberately executed the code (by calling the INT *n* instruction, for example), the code is a pseudo-interrupt handler. If a program accidentally executed the code (by loading an invalid segment descriptor and causing a GP fault, for example), the code is a trap handler.

For the rest of this chapter, the term exception handler refers to both trap handlers and pseudo-interrupt handlers.

Exception handlers are installed using the TrapHandler operations, SetTrapHandler, SetDefaultTrapHandler, and Set386TrapHandler. Each of these operations is described in the *CTOS/Open Application Programming Interface Specification*. Most set a local exception handler, which only traps those faults and/or pseudo-interrupts generated in its partition. SetDefaultTrapHandler, however, establishes a global trap handler which processes all traps of a given type, no matter what their source.

State of the System on Entry to an Exception Handler

On entry, exception handlers encounter conditions similar to those for raw interrupt handlers, but with two important differences.

First, on entry to an exception handler interrupts are enabled. This means that exception handlers can make calls to the operating system. It also means that exception handlers must be reentrant, because the same trap or pseudo-interrupt could occur while the exception handler is processing a previous one.

Second, the contents of all registers other than CS and IP are undefined, and must be saved before use. Exception handlers use the interrupted program's stack. The operating system does not provide a stack for their use. Note also that while the operating system sets up DS for a raw interrupt handler, it does not do so for an exception handler. This requires virtually all exception handlers to contain at least some assembly language.

System Pseudo-Interrupt Handlers

Certain commonly-used exceptions have additional support in the operating system. For example, if a program needs to use the programmable interval timer (PIT), it sets up a pseudo-interrupt handler to process timer pseudo-interrupts.

The entry conditions for a system pseudo-interrupt handler are identical to those for a mediated hardware interrupt handler.

Writing Raw Interrupt Handlers

The following conditions apply to all raw interrupt handlers:

1. In real mode, the interrupt handler must save all registers before using them. In protected mode, this requirement does not apply.

2. On entry, DS is set to the value requested on the SetIntHandler call. The contents of ES are undefined.

3. On entry, the stack may have either of two values.

   In real mode, the interrupt handler uses the interrupted process's stack. It should therefore keep its stack use to an absolute minimum.

   In protected mode, the interrupt handler uses one of the operating system's stacks, but its stack use should still be kept low.

   Note that in a raw interrupt handler, SS never equals DS.

4. On entry, interrupts are disabled. They must remain disabled throughout the execution of the interrupt handler.

5. The interrupt handler can make no operating system calls other than MediateIntHandler, which converts the raw interrupt handler to a mediated interrupt handler.

6. The interrupt handler must send a Non-Specific End-of-Interrupt signal to one or both programmable interrupt controllers. It must also perform any end-of-interrupt processing required by the device it serves.

7. Raw interrupt handlers must restore any saved registers, then exit using the IRET instruction.

8. In protected mode, the IRET instruction must be followed immediately by a JMP to the beginning of the interrupt handler.

It is rarely advisable to write a raw interrupt handler. The performance improvement, especially in protected mode, can be negligible and the difficulty in writing one can be substantial. When you need a high-performance interrupt handler, you should usually use a mediated interrupt handler written in assembly language.

If a particular situation absolutely requires the use of a raw interrupt handler, the best strategy is usually first to perform the work that required the handler to be raw, then to call MediateIntHandler. This forces the operating system to reset the programmable interrupt controller when the interrupt handler exits, freeing the programmer from responsibility for this hardware-dependent task.

Writing Mediated Interrupt Handlers

Mediated interrupt handlers are the preferred method for processing hardware interrupts. In protected mode, the overhead imposed by using a mediated interrupt handler instead of a raw one is approximately two machine instructions. In real mode, the overhead is somewhat larger.

The following conditions apply to all mediated interrupt handlers:

1. The operating system saves all registers before passing control to the interrupt handler.

2. On entry, DS is set to the value requested on the SetIntHandler call. The contents of ES are undefined.

3. The interrupt handler uses one of the operating system's stacks, but its stack use should still be kept low. Note that SS never equals DS.

4. On entry, interrupts are enabled. They should generally remain enabled throughout the execution of the interrupt handler, but they can be disabled briefly to protect critical sections.

5. The interrupt handler can make operating system calls. It should usually restrict itself to PSend, SetTimerInt and ResetTimerInt, however.

6. If the interrupt handler was installed with fDeviceInt equal to TRUE (0xFF), the operating system performs the end-of-interrupt processing for the programmable interrupt controller.

7. The interrupt handler must perform any end-of-interrupt processing required by the device it serves.

8. Mediated interrupt handlers should exit using the RET instruction. The operating system restores the previous contents of all registers after the interrupt handler exits.

When writing a mediated interrupt handler in a high level language, you can usually just make the interrupt handler a procedure that takes no parameters, and returns nothing to the caller.

However, in some models of compilation (particularly the Medium model), some compilers may assume that DS and SS are equal. Acting on that assumption usually causes a protection fault. If this problem occurs, you can usually remedy it by changing the way the program references the offending variable. See your compiler documentation for more information on the compiler you use.

Writing System Pseudo-Interrupt Handlers

All system pseudo-interrupt handlers are of the mediated type. See the preceding section for information about mediated interrupt handlers in general. For an example of a system pseudo-interrupt handler, see Listing 15-7 "A Timer Interrupt Handler."

Writing Exception Handlers

Exception handlers are usually used to invoke a cleanup routine when a program encounters a fatal error, such as a GP fault. For example, the CTOS Editor uses an exception handler to save its recovery typescript file when it encounters a GP fault. See "Trapping Protection Faults" in Chapter 3 for an example of a GP fault handler.

The following conditions apply to all exception handlers:

1. The exception handler must always save all registers before using them. This restriction applies in both real and protected mode.

2. On entry, the contents of all registers except CS and IP are undefined.

3. On entry, the stack segment and stack pointer are those of the interrupted program. Exception handlers use the stack of the interrupted program, and should therefore keep their stack use to an absolute minimum.

4. On entry, interrupts are enabled. The exception handler can disable them briefly to protect critical sections.

5. The exception handler can make operating system calls, but care should be taken not to use too much stack space.

6. There is no end-of-interrupt signal associated with a trap. Trap handlers do not need to perform any special tasks to indicate completion of their work.

7. Exception handlers must restore any saved registers, then exit using the IRET instruction.

8. Unlike raw interrupt handlers, exception handlers do not need to follow the IRET with a JMP.

When an exception handler is contained within a program, all these restrictions still apply, but the programmer has more control over the environment the exception handler inherits. For example, if an exception handler uses a lot of stack space, the programmer can increase the program's stack size so that the exception handler is sure to have enough.

Listing 19-1 shows a useful trick to use when exception handlers are contained within a program. The assembly language part of the exception handler is just a stub which saves all registers and then transfers control to a high-level language procedure, MyTrapHandler. That procedure can then do whatever it wants. Remember, though, that exception handlers should be reentrant.

```
;A trap handler.  Needs underbars for C language.
_asmTrapHandler segment word 'code'
assume cs: _asmTrapHandler
public asmTrapHandler
asmTrapHandler proc far
    pusha
    push ds
    push es
    mov  ax,DGROUP
    mov  ds,ax
    call TrapHandler
    pop  es
    pop  ds
    popa
    iret
asmTrapHandler endp
_asmTrapHandler ends
```

Listing 19–1. An Assembly Language Stub for an Exception Handler

Writing 80386 Exception Handlers

The Set386TrapHandler creates an 80386-only exception handler. The normal style of exception handler works on an 80386 processor, but an 80386-only exception handler has access to more information about the interrupted program.

A program should never install an 80386 exception handler unless it is running on an 80386 processor. Any program that uses an 80386 exception handler should be labelled as running only on 80386-based workstations.

An 80386 exception handler is almost identical to a standard (80286) one. The difference is that while the operating system pushes 16-bit values for the interrupted program's CS, IP and flags on the stack for a standard exception handler, it pushes 32-bit values for an 80386 exception handler. This allows the exception handler to gain access to the upper 16 bits of the 80386 flags register.

When you write an 80386 exception handler, you must make sure that the handler executes a 32-bit return, instead of the normal 16-bit return. Otherwise, the operating system will pop only half of the interrupted program's CS, IP and flags information, leaving three extra words on the program's stack. This will cause the program to fail at some later point in its execution.

Listing 19-2 shows two assembly-language stubs for an 80386 exception handler, one for an assembler that understands 80386 instructions, the other for an assembler that does not.

```
;386 trap handlers.  Underbars are for C language
_asmTrapHandler386 segment word 'code'
assume cs: _asmTrapHandler386

public asmTrapHandler386
asmTrapHandler386 proc far

    pusha
    push ds
    push es
    mov  ax,DGROUP
    mov  ds,ax
    call TrapHandler386
    pop  es
```

continued...

```
    pop  ds
    popa
;   Only code one of the two choices below
;   1. Use the 386 32-bit return instruction, or
    iretd

;   2. Set an instruction prefix to force a 32-bit return
    db   66h
    iret

;   Normal segment and procedure end stuff
asmTrapHandler386 endp
_asmTrapHandler386 ends
```

Listing 19–2. An Assembly Language Stub for an 80386 Exception Handler

20

Inter-Module Communication

This chapter discusses low-level communication with workstation add-on modules. These modules attach to the workstation expansion bus, called the X-Bus. The topics addressed in this chapter are of use primarily to people who build X-Bus modules, but may also be of interest to those who want to write drivers for existing modules.

This chapter assumes that you are familiar with the NGEN workstation architecture, and with basic X-Bus concepts. For more information on these topics, see your hardware and operating system documentation.

Introduction

In general, the only programs that communicate directly with X-Bus modules are the system services that manage those modules. For example, the video portion of the operating system manages all communication between application programs and a workstation's graphics module, and Voice/Data Services manages all communications between applications and a Voice Processor module.

CTOS provides several mechanisms to assist the builders of X-Bus modules in writing the system software that manages those modules. The three most commonly used ones are:

- the X-Bus I/O mechanism
- the X-Bus memory window mechanism
- the X-Bus interrupt mechanism

The X-Bus I/O mechanism assigns a range of I/O addresses to each X-Bus module. A program running on the main processor can perform I/O to a module by reading from (or writing to) the appropriate range of addresses.

The X-Bus memory window mechanism allows a program to map an X-Bus module's memory address space to a portion of the main processor's address space. The program can then access this mapped X-Bus module memory as if it were physically part of the processor's main memory. When the program reads or writes to this mapped memory, the operating system updates the X-Bus module's memory accordingly, and does so in a way that is transparent to the application.

The X-Bus interrupt mechanism allows an X-Bus module to notify the main processor that it needs attention. The processor can then transfer control to a custom-written interrupt handler that performs the needed work.

The X-Bus also supports DMA transfers between X-Bus modules and main processor memory, and provides a bus-mastering capability for intelligent X-Bus modules. However, a discussion of these topics is beyond the scope of this chapter.

More extensive documentation on X-Bus interface software and on X-Bus hardware module development is available to X-Bus licensees. Contact Unisys Network Computing Group for more information about X-Bus licensing.

Information Required

Before a program can communicate with an X-Bus module at all, its programmer must have some basic information about that module. The programmer needs:

1. The type code assigned to that X-Bus module.

   These one-byte codes are assigned by CTOS/BTOS technical support, and are unique to each type of hardware module. See your operating system documentation for a list of currently-defined module type codes. Or, see the documentation for a particular module to determine its type code.

2. A list of valid I/O addresses for the module (between 0 and 255)

   Each type of module may use any or all of the 256 possible I/O addresses for an X-Bus module. These addresses are assigned by the designers of each module. See the documentation for individual modules to determine which addresses are used by that module, and for what purpose.

3. The size of the X-Bus memory window supported by the module, if any.

   X-Bus windows come in several sizes. Which size a module uses is determined by the designers of that module. The possible X-Bus window sizes are 96K, 224K, and 480K.

These three items of information allow the programmer to establish communication with the module. What type of communication or programming is appropriate for the module depends entirely on its intended use and on its hardware design.

Performing I/O to X-Bus Modules

Performing I/O is the simplest method of communicating with an X-Bus module. It is also a useful mechanism for transferring small chunks of information between a module and its driver program.

The X-Bus I/O Base Address

Each X-Bus module has 256 I/O port addresses available to it. A program can address any one of these addresses by adding the desired port address number (0 - 255) with an *I/O base address* for the module. These two numbers together uniquely specify the port address which the program wants to access.

The I/O base address is a number that defines the range of addresses for an X-Bus module in a given position on the X-Bus. I/O base addresses are assigned to modules from left to right, starting at 256 (100h) and incrementing by 256 (100h). So, the first module (immediately to the right of the main processor) has an I/O base address of 100h, the next module has an I/O base address of 200h, and so on.

So, if a program wants to read port 6 from the second module on the X-Bus, it reads port 206h.

You may wonder at this point how the program discovers the position of the module it needs to address. The operating system provides a mechanism, QueryModulePosition, that automatically obtains the position of a given module type.

Determining a Module's I/O Base Address

To determine the base address for an X-Bus module in a particular system, the programmer must first have a way to find that module's position on the X-Bus.

The following code fragment gets the position of an imaginary Foo module using the Foo module's type code, then calculates the I/O base address for the module.

```
/* Find the position of a module with the right type code
*/
erc = QueryModulePosition(&iPosition, 2, FOOTYPE, 0);
if(erc == 35) { /* no such module */
    printf("\nNo Foo module found.");
    ErrorExit(erc);
    }
else
    CheckErc(erc);
/* figure the base address */
iBaseAddress = (iPosition-1) * 0x100;
```

Listing 20–1. Calculating an X-Bus Module's Base Address

Performing the I/O

Once the I/O base address for a module for a module has been calculated, the program can perform I/O to the module using the processor's IN and OUT instructions. The I/O port number used should be the module's base address plus the desired I/O port number. The following code fragment shows an example of reading from the Foo module's I/O space.

```
/* Read a byte from the X-Bus module by adding the module
   port number to the base address
*/
bWhatIsIt = LockIn(iBaseAddress+iXBPortIWant);
```

Listing 20–2. Performing I/O to an X-Bus Module

Accessing Memory in X-Bus Modules

Accessing memory in X-Bus modules via an X-Bus window allows a program to transfer large amounts of data to and from the module quickly, with a minimum of effort.

Of course, the module must be designed to support an X-Bus window before a program can use one with it.

Setting Up an X-Bus Window

Setting up an X-Bus window requires that the programmer know three things: the module type code, the window size supported by the module, and the I/O port which controls the module's window mapping register.

The operating system uses the module's window mapping register to make that module's window uniquely addressable on the X-Bus. Which port ID in an X-Bus module controls its window mapping register is determined by the module's designers.

The following code fragment establishes an X-Bus window. It assumes that the type code, window size, and window mapping control port have been previously defined.

```
/* array of pointers to 64K chunks of X-Bus window
   ie 8 64K chunks (512K total) for a 480K window.
*/
Pointer rgpWindowSegs[CWINDOWSEGS];
Word    srgpWindowSegs = CWINDOWSEGS * sizeof(Pointer);
/* wEAR is top bits of X-Bus window address.  Only needed
   for ISRs.
*/
Word    wEAR = 0;
```

continued...

```
    .
    .
    .
/* Establish an X-Bus window with our Foo module */
CheckErc(MapXBusWindowLarge(FOOTYPE, 1, &rgpWindowSegs,
        srgpWindowSegs, &cWindowSegsRet, FOOWINDOWSIZE,
        &wEAR, FOOWINDOWMAPPORT));
if(cWindowSegsRet != CWINDOWSEGS) {
    printf("\nUnexpected number of window segments");
    exit();
    }
```

Listing 20–3. Setting Up an X-Bus Window

Reading and Writing to Module Memory

Once the X-Bus window has been set up, that memory can be read and written as if it resided in main processor memory. For example, if the window was initialized as shown in the example above, the program could then execute the following instructions:

```
pFirstWindowSeg = rgpWindowSegs[0];
/* store a value in X-Bus memory */
*pFirstWindowSeg = 5;
```

Servicing X-Bus Module Interrupts

Many X-Bus modules incorporate microprocessors into their design, allowing them to perform certain functions independently of the main processor. These modules usually communicate with their control software on the main processor by means of interrupts.

Three X-Bus interrupts are available: XINT0, XINT1, and XINT4. XINT0 and XINT1 are usually used by disk drive modules. The one most commonly used by third-party module designers is XINT4. This interrupt is multiplexed by the operating system among all programs that request to service it. This allows any number of modules to use XINT4 on the same computer system.

The XINT4 mechanism requires a system service called the X-Bus Interface (XBIF) service. The XBIF service implements a round-robin polling scheme for all XINT4 interrupt handlers. It also performs additional, operating-system-related X-Bus management services.

The Elements of Your Program

Generally, a program that makes use of the X-Bus interrupt mechanism must do three things:

1. Initialize the module it controls and install its interrupt handler.
2. Handle the X-Bus interrupts generated by the module, using an X-Bus interrupt handler.
3. Perform the work implied by each interrupt.

Item one should be performed in the program's initialization sequence.

Item two should be performed by a customized interrupt handler, written for that module.

Item three constitutes the program's main work.

Establishing Communication with an X-Bus Module

Modules that have enough intelligence to generate X-Bus interrupts usually are also able to access memory on the main processor. This section assumes that is the case.

The initialization process for modules that cannot access main processor memory is likely to be similar to the one presented here. The main difference would be that any control structures would reside in the module's memory or I/O space.

The operating system provides an area of memory that programs can use when initializing the modules they control. This area of memory, called the X-Bus Interface Structure (XBIS), is a 16-byte structure of undefined contents. A program can gain exclusive access to this structure by calling the LockXbis procedure. The XBIS is always located at physical address 0:0400h, so that modules can know in advance where to read from during initialization.

In general, the steps to initialize an intelligent module using the XBIS are:

1. Allocate a control structure in the program's memory, which will be used for communication with the X-Bus module.

2. Install the ISR that will serve the module's interrupts, using SetXbusMISR.

3. Call LockXbis to reserve the XBIS structure. LockXbis returns a pointer to the XBIS structure, if it is not currently in use.

4. Place the type code of the module in byte 0 of the XBIS.

5. Call PaFromP to get the physical address of the control structure, then place the lower-order 24 bits of the physical address in bytes one to three of the XBIS.

6. Place any other needed information in the XBIS.

7. Output an I/O command to the module, telling it to read the contents of the XBIS. (The module must know the location of the XBIS and what to do with its contents. This is up to the module designers.)

8. Query the module's status, and unlock the XBIS if the module read its contents successfully.

After completion of these steps, the program should be able to communicate with the module through the shared control structure.

The X-Bus Interrupt Handler

The multiplexed X-Bus interrupt mechanism works in a round-robin fashion. Each installed X-Bus interrupt handler is called in turn, until one of them identifies the interrupt as belonging to its module.

All X-Bus multiplexed interrupt handlers should return a boolean status code. If the current X-Bus interrupt was generated by the module it controls, the interrupt handler should process the interrupt and return TRUE (0xFF). If the current interrupt was generated by another module, the interrupt handler should do nothing, and return FALSE (0x00).

The module must implement a notification mechanism so that its ISR can determine whether that module generated the interrupt. One common method is for the module to set or clear a byte in the shared control area just prior to generating the interrupt. The ISR can then check the status of this byte to see if its module generated the interrupt.

Note that when this method is used, it is possible that the module set the status byte and intended to generate an interrupt, but another module generated an X-Bus interrupt first. This can result in a race condition if the interrupt handler is improperly designed. The fragment shown in Listing 20-1 correctly handles this potential exception.

Listing 20-1 shows a code fragment which contains the module-independent portion of an X-Bus interrupt handler.

```
Isr_Code SEGMENT PUBLIC 'CODE'

PUBLIC Isr
Isr PROC FAR

ASSUME CS: Isr_Code
ASSUME ES: NOTHING
ASSUME DS: DGroup
ASSUME SS: NOTHING

    MOV CL, 0      ; RETURN FALSE if not for us.

; see if module set byte in control structure indicating
; it generated an interrupt
    MOV BL, 0FFh
LOCK XCHG BL, Cstruct_intStatus
    CMP BL, 0FFh
    JE IsrRet     ; not for us if status is FF

; If it's for us, perform module-specific work

    <Module-specific processing>
        .
        .
        .

; Wake up main program when we're done with module.
;
DoWakeup:
    PUSH exchange
    LEA  AX, IsrMsg
```

continued...

```
    PUSH DS
    PUSH AX
    CALL PSend
    MOV  CL, 0FFh                ; RETURN TRUE

IsrRet:
;
; turn off module interrupt and return.  Clear isr
; even if not for us just in case
;
    MOV DX, IoPort
    IN AL, DX
    MOV AX, CX
    RET

Isr ENDP
Isr_Code ENDS

END
```

Listing 20–4. Skeleton of an X-Bus Multiplexed Interrupt Service Routine

Appendixes

A

Accessing CTOS Operations From Assembly Language

This appendix describes (1) accessing CTOS operations from programs written in assembly language and (2) the conventions for argument-passing, register usage, and segments, classes, and groups. Assembly language examples illustrate both operating system access and the conventions.

Argument Passing

The operating system and object module procedures (such as byte streams) deal with data items and structures of many different sizes ranging from single-byte items, such as Boolean flags, to multibyte structures, such as request blocks and Byte Stream Work Areas.

Three of these are special: 1-byte, 2-byte, and 4-byte data items. Only these are passed as arguments on the stack or returned as results in the registers.

When it is necessary to pass a data structure as an argument, the 4-byte logical memory address of (pointer to) the data structure is used as the argument.

Note that pointers are arranged in memory with the low-order part, the offset, at the lower memory address and the high-order part, the segment base, at the higher memory address. However, the processor architecture of the workstation is such that stacks grow from high memory addresses toward low memory addresses. Hence, the high-order part of a pointer is pushed before the low-order part.

Also note that byte arguments are pushed on to the stack as words, with the low-order byte of the word being the argument.

If the argument is Boolean, the convention is to use a byte value of 0FFh for true and 0 for false. This is not simply nonzero or 0, since the actual test used is to see if the least significant bit is set or clear.

Register Usage

The contents of the registers for the current code segment (CS), data segment (DS), stack segment (SS), stack pointer (SP), and base pointer (BP) are preserved across calls: they are the same on return as they were just prior to the pushing of the first argument onto the stack. It is assumed that SS and SP point, respectively, to the base of the stack and the top of the stack, and this stack is used, in general, by the called service. None of the other registers or the flags is preserved across calls.

If the procedure called is a function (that is, if it returns a value), the return value is placed in one or more registers, according to the type of value.

If the value is a byte, it is returned in AL (the low byte of register AX).

If the value is a word (two bytes), it is returned in AX (most of the object module procedures return a value of type *ercType*, which is actually a word).

If the value is a doubleword (a pointer or a logical file address), the most significant word (or segment part of a pointer) is returned in ES and the least significant word (or offset of a pointer) is returned in BX.

BP Register

The above conventions place no particular requirement on the contents of BP. However, the Debugger cannot trace the stack of a procedure being debugged if BP is not used according to one of the two standard BP usage conventions.

The most common convention uses BP as a pointer to the stack, just above the stack space assigned for local variables. All references to arguments and local variables are then made relative to BP. References to parameters have a positive offset from BP. References to local variables have a negative offset from BP. Each procedure has a prologue and an epilogue that looks like the following:

```
PUSH      BP
MOV       BP, SP
SUB       SP, sLocalFrame
  .
  .
  .
MOV       SP,BP
POP       BP
RET       sArgumentFrame
```

The values for sLocalFrame and sArgumentFrame are always even. sLocalFrame is the number of bytes of local variables, and sArgumentFrame is the number of bytes pushed on to the stack by the calling service (each PUSH instruction adds two bytes).

The other convention also uses BP as a pointer to the stack, but just *below* the stack space assigned for local variables. All references to arguments and local variables are then made as positive offsets to BP. Each procedure has a prologue and an epilogue that looks like the following:

```
SUB       SP, sLocalFrame
PUSH      BP
MOV       BP, SP
  .
  .
  .
POP       BP
RET       sArgumentFrame + sLocalFrame
```

Segments, Classes, and Groups

Object module procedures assume that the registers SS and DS are set to DGroup, because all the object modules are compiled as medium model. The registers are set automatically for a high-level language main program, when it uses the medium model. However, an assembly language main program must explicitly set the registers. The program should include code similar to the following:

```
EXTRN Exit
PUBLIC Main

Stack SEGMENT STACK 'Stack'
DB sStack DUP (?)
raStackLim LABEL BYTE
Stack ENDS

DGroup GROUP Stack

YourCode SEGMENT
ASSUME CS:YourCode
Main PROC FAR

     MOV       AX, DGroup
     MOV       DS, AX
     MOV       SS, AX
     MOV       SP, OFFSET DGroup:raStackLim
.
.
.
     CALL      Exit
Main ENDP
YourCode ENDS
END Main
```

The segments included in the group DGroup are

Segment	Class
Const	Const
Statics	Const
Data	Data
Stack	Stack
Memory	Memory

Example Program

The TypeSector program copies the first sector of a file to the video display, using CTOS file system operations to open and read the file and SAM (the Sequential Access Method) to write to the video display. The file specification used is obtained from the Executive. The program assumes the file name is specified in a form like

```
Command  TypeSector
   TypeSector
    File name        Sample.File
```

The TypeSector program calls ErrorExit and returns to the Executive if an error is detected.

The program consists of two modules, TypeSector and TypeArg. The modules are assembled and linked as follows:

```
Command  Assemble
   Assemble
    Source files     TypeSector.Asm
                     .
                     .
                     .
```

```
Command  Assemble
   Assemble
     Source files     TypeArg.Asm
                      .
                      .
                      .

Command  Link
   Link
     Object modules   TypeSector.Obj TypeArg.Obj
     Run file         TypeSector.Run
                      .
                      .
                      .
```

```
;TypeSector.Asm

;Public and external declarations.

Public Main
EXTRN RgParam:FAR, OpenFile:Far, Read:FAR
EXTRN WriteByte:FAR, WriteBSRecord:FAR, BSVid:Byte
EXTRN TypeArg:FAR, CloseFile:Far
EXTRN Exit:FAR, ErrorExit:FAR

;Segment declarations.
;
;All Segments used are mentioned in the order they
;are linked.

TypeCode SEGMENT PUBLIC 'Code'
TypeCode ENDS

Const SEGMENT PUBLIC 'Const'
Const ENDS

Data SEGMENT PUBLIC 'Data'
Data ENDS

Stack SEGMENT STACK 'Stack'
Stack ENDS
```

```
;Group the segments together for compatibility
;with Convergent object modules.

DGroup GROUP Const, Data, Stack

;Stack declaration.

;Declare 200h bytes (100h words) in this module.
;See the  Linker Librarian Manual for combining
;stack segments in different modules.
;raStackLim is placed so that the stack is the
;size of the sum of all stack declarations.

Stack SEGMENT
DW 100h DUP (?)
raStackLim LABEL BYTE
Stack ENDS

;Data declarations.

;All of the variables used in this module are
;declared here.

Data SEGMENT
sDataRet DW ?                   ; This is the variable
                                ; that the the CTOS Read
                                ; calls and the byte
                                ; stream WriteBSRecord
                                ; uses to fill in the
                                ; actual count of bytes
                                ; read.

sdRet DB 6 DUP (?)              ; This is the structure
                                ; used to obtain
                                ; pararmeters from the
                                ; Executive.  The sdRet
                                ; structure is defined as
                                ; a pointer (four bytes)
                                ; followed by a count
                                ; (two bytes.)
```

```
fh DW ?                          ; File handle for the
                                 ; source file

EVEN
rgBuf DB 512 Dup (?)             ; Word-aligned input
                                 ; buffer.

DATA ENDS

;Macro definition for checking errors.

; A procedure of ErcType returns the erc in
; register AX.  If AX is nonzero then simply call
; ErrorExit.

$SAVE NOGEN
%*DEFINE (CheckErc)LOCAL ok (
     AND                         AX, AX
     JE                          %ok
     PUSH                        AX
     CALL                        ErrorExit

%ok:
)
%RESTORE

;Main code segment follows.

TypeCode SEGMENT
ASSUME CS: TypeCode
Main PROC FAR
```

```
; Initialization.

; Set the segment registers (SS, DS) and stack.
; Since the CPU chip disables interrupts for one
; instruction following a move to a segment
; register, there is no problem initializing the
; stack pointer (SP).
;
; Since the segment registers SS and DS are being
; initialized to DGroup, DGroup must be explicitly
; specified when referring to the offset of a
; variable.  If this is not done, then the offset
; of a variable is from the start of the segment
; in which it is declared, not from the start of
; the group of segments.

    MOV         AX, DGroup      ; Set SS
    MOV         SS, AX
ASSUME SS:DGroup
    MOV         SP, OFFSET DGroup:raStackLim
    mov         BP, SP          ; Set BP for
                                ; compatibility with
                                ; Convergent object
                                ; modules.
    PUSH        SS
    POP         DS              ; Set DS.
ASSUME DS:DGroup
;
; Type the parameter to the video display using
; TypeArg.
;
; erc := TypeArg (iParam, jParam);

    MOV         AX, 1           ; iParam (1).
    PUSH        AX
    XOR         AX, AX          ; jParam (0).
    PUSH        AX
    CALL        TypeArg
    %CheckErc
```

```
; Type a ":" and a new line character.
;
; erc := WriteByte (pBSWA,B);
;
    PUSH        DS              ; pBSWA (pBSVid).
    MOV         AX, OFFSET DGroup:bsVid
    PUSH        AX
    MOV         AL, ':'         ; B (:).
    PUSH        AX
    CALL        WriteByte
    %CheckErc
    PUSH        DS              ; pBSWA (pBsVid).
    MOV         AX, OFFSET DGroup:bsVid
    PUSH        AX
    MOV         AL, 0Ah         ; b (new line).
    PUSH        AX
    CALL        WriteByte
    %CheckErc

; Get the file name from the Executive.
; (parameter 1, subparameter 0)
;
; erc := RgParam (iParam, jParam, pSdRet)

    MOV         AX, 1           ; iParam (1).
    PUSH        AX
    XOR         AX, AX          ; jParam (0).
    PUSH        AX
    PUSH        DS              ; pSdRet.
    MOV         AX, OFFSET DGroup:SdRet
    PUSH        AX
    CALL        RgParam
    %CheckErc

; Open the file for mode read.
;
; erc := OpenFile (pFh, pbFileSpec, cbFileSpec,
;         pbPassword, cbPassword, mode);
```

```
    PUSH        DS              ; pFh
    MOV         AX, OFFSET DGroup:fh
    PUSH        AX
    PUSH        WORD PTR sdRet + 2 ; pbFileSpec
    PUSH        WORD PTR sdRet
    PUSH        WORD PTR sdRet + 4 ; cbFileSpec
    XOR         AX, AX          ; pbPassword (null).
    PUSH        AX
    PUSH        AX
    PUSH        AX              ; cbPassword (0).
    MOV         AX,'mr'         ; modeRead
    PUSH        AX
    CALL        OpenFile
    %CheckErc

;Read in the first sector (512 bytes).
;
; erc := Read (fh, pBufferRet, sBufferMax,
;              lfa, psDataREt);

    PUSH        fh              ; fh.
    PUSH        DS              ; pBufferRet.
    MOV         AX, OFFSET DGroup:rgBuf
    PUSH        AX
    MOV         AX, 512         ; sBufferMax.
    PUSH        AX
    XOR         AX, AX          ; lfa (0).
    PUSH        AX
    PUSH        AX
    PUSH        DS              ; psDataRet.
    MOV         AX, OFFSET DGroup:sDataRet
    PUSH        AX
    CALL        Read
    %CheckErc
```

```
;Write the buffer to the video display.
;
; erc := WriteBsRecord (pBSWA, pb, cb, pcbRet);

    PUSH        DS              ; pBSWA (pBsVid).
    MOV         AX, OFFSET DGroup:bsVid
    PUSH        AX
    PUSH        DS
    MOV         AX, OFFSET DGroup:rgBuf
    PUSH        AX
    PUSH        sDataRet        ;cb
    MOV         AX, OFFSET DGroup:sDataRet
    PUSH        DS
    PUSH        AX
    CALL        WriteBSRecord
    %CheckErc

; Return to the Executive.
;
     CALL       Exit

Main ENDP                       ; End of Main.

TypeCode ENDS                   ; End of segment.

END Main                        ; End of module (specify
                                ;starting point as Main.)
```

The TypeArg procedure types a parameter passed from the Executive to the video display. It is called with two parameters, iParam and jParam, and is of the type ercType. It can be called from C as follows:

```
erc = TypeArg (iParam, jParam);
```

The procedure returns 0 if no errors were encountered; otherwise it returns the error in register AX.

The procedure is reentrant and uses no static variables.

This is not a main program but a procedure. It is assumed that the segment registers are properly set before calling this procedure, with DS and SS set to DGroup.

```
;TypeArg.Asm

; Public and external declarations.

PUBLIC TypeArg
EXTRN RgParam:FAR, WriteBSRecord:FAR, BsVid:BYTE

; The procedure uses 18 bytes of stack for itself,
; not counting calls to other procedures, as
; follows:
;
;   four bytes for parameters passed to it,
;   four bytes for the return address of the
;    calling service
;   two bytes to store the BP of the calling
;    service
;   eight bytes of local variables.

Stack SEGMENT STACK 'Stack'
DB 18 DUP (?)
Stack ENDS
DGroup GROUP Stack

TypeArgCode SEGMENT
ASSUME CS:TypeArgCode, DS:DGroup, ES:NOTHING, SS:Dgroup
TypeArg PROC FAR

; Set the local variables and parameters as EQUs.

SArgFrame EQU 4                 ; Parameters to argument
                                ; are two words (four
                                ; bytes).
SLocalFrame EQU 8               ; Eight bytes of local
                                ; variables.

    PUSH        BP              ; Save the calling
                                ; service's BP.
    MOV         BP, SP          ; Use BP as a frame
                                ; pointer.
    SUB         SP, SLocalFrame
```

```
iParam EQU WORD PTR [BP + 8]
                                ; First parameter on
                                ; stack.

jParam EQU WORD PTR [BP + 6]
                                ; Second parameter on
                                ; stack

sdRet EQU BYTE PTR [BP - 6]
pbArg EQU DWORD PTR [BP - 6]
cbArg EQU WORD PTR [BP - 2]
                                ; sdRet is a 6 byte
                                ; structure consisting
                                ; of a pointer (pbArg;
                                ; four bytes) and a count
                                ; (cbArg; two bytes)
                                ; located on the stack at
                                ; SS:[BP - 6] to
                                ; SS:[BP - 2] .

sDataRet EQU WORD PTR [BP - 8]
                                ; sDataRet is the count
                                ; of bytes actually
                                ; written to the video
                                ; display, and is
                                ; ignored in this
                                ; procedure.

; erc := RgParam (iParam, jParam, pSdRet);

    PUSH        iParam          ; iParam.
    PUSH        jParam          ; jParam.
    LEA         AX, sdRet       ; pSdRet.
    PUSH        SS
    PUSH        AX
    CALL        RgParam
    AND         AX, AX          ; Check for errors.
    JNZ         Finish
```

```
; erc := WriteBsRecord (pBSWA, pb, cb, pcbRet);

    PUSH        DS                  ; pBSWA (pBSWA (pBsVid).
    MOV         AX, OFFSET DGroup:bsVid
    PUSH        AX
    LES         AX, pbArg           ; pb.
    PUSH        ES
    PUSH        AX
    PUSH        cbArg               ; cb.
    LEA         AX, sDataRet        ; pcbRet.
    PUSH        SS
    PUSH        AX
    CALL        WriteBsRecord

; All done, so return erc in AX.

Finish:

    MOV         SP,BP               ; Remove the local
                                    ; variables from the
                                    ; stack.
    POP         BP                  ; Restore the BP of the
                                    ; calling service.
    RET         sArgFrame           ; Return with arguments
                                    ; (four bytes) removed
                                    ; from the stack.

TypeArg ENDP
TypeArgCode ENDS
END
```

B

NLS Templates

The currently defined formats and default templates, along with a description, examples, and maximum length, are listed in Figure B-1. The maximum lengths given refer to the length of the actual date/time string produced by NlsFormatDateTime, not the format string.

NOTE: When changing the templates and the various strings used to construct the dates, do not exceed the maximum string length. Allow for the maximum length when using one of the defined formats.

The following rules summarize how NlsFormatDateTime interprets the templates below:

1. Text contained within exclamation marks (!) gets expanded according to the date and time as passed to NlsStdFormatDateTime. Other text is reproduced verbatim.

2. Letters expand to attributes of the specified date and time, as follows:

Letter	Date and Time
a:	am, pm, noon, or midnight
d:	day of the month (numeric)
h:	hour (numeric, 12 hour clock)
m:	minute (numeric)
n:	month of year (alphabetic)
o:	month of year (numeric)
s:	second (numeric)
t:	hour (numeric, 24-hour clock)
w:	day of week (alphabetic)
y:	year (numeric)

The actual letters may be selected via the NLS data tables. This provides the ability for application programs to accept template specifications from users in a natural fashion based on country.

3. The attributes expand to as many character positions as are held by adjacent identical letters. For example, !yyyy! expands to "1985" and !yy! expands to "85". Left truncation is used for numeric fields; right truncation is used for alphabetic fields. By default, template fields that are longer than the attribute filled in are padded with spaces. Numbers are padded to the left; strings to the right.

4. Template letters that are capitalized result in capitalization of the corresponding letters in the expanded string. For example, !AA! expands to "AM", !aa! expands to "am", and !Aa! expands to "Am".

5. An asterisk (*) before a template letter defines a variable format repetition. The asterisk causes the next format character to be used repetitively until the entire expansion is complete. For example, !*d! results in "1" or "12". Also, !*n! results in "september" while !*N! results in "SEPTEMBER".

6. An asterisk may be preceded by a series of like format characters (case may differ). !N*n! is an example. This is useful in controlling the case of the leading character in variable length fields. The example, !N*n!, would result in "September".

7. Zeroes before a template letter result in leading zeroes preceding the significant part of the number. For example, !0d! results in "09" or "12".

8. The first numeric digit (and first character) within a template field, such as the digit "2" in !2*w!, causes the selection of a date name table. In this case, date name table number 2 is selected. This selection remains in effect until a new table is selected or the date/name string is complete.

   This type of template is used, for example, to select alternate spellings for the case where the abbreviation for a month or day name is not merely the left-most letters of the full name. A leading 0 is not a table identifier. A template with a leading 0 that is to result in a string is illegal.

Templates are processed left to right. The first table (number 1) is assumed at the start of processing for a template. If a change of table number is found within a template, it takes effect immediately and remains in effect until changed or until the end of the template is reached. Appearance of a numeric digit within a template field has no effect on formatting. If a template calls for a date name table that does not exist, the message, "Invalid template index," is returned to the user.

A small set of options is standardized to handle the varying requirements of application programs for different types of date and time formats. These date and time formats are selected by an ID code. ID codes 0 through 32767 are reserved; 32768 through 65535 are available for definition.

Description	Identifier	Default Templates	Examples	Maximum Length
Columnar numeric date and time of day	0	!oo!/!0d!/!0y! !hh!:!0m!!2*A!	7/18/85 9:03AM 10/08/85 11:13PM	16
Numeric date and time of day	1	!*o!/!0d!/!0y! !*h!:!0m!!2*A!	7/18/85 9:03AM 10/08/85 11:13PM	16
Columnar date and time of day	2	!Nnn! !dd! !yyyy! !hh!:!0m! !2*A!	Jul 1, 1985 9:03 AM Jul 18, 1985 11:13 PM	26
Date and time of day	3	!Nnn! !*d!, !yyyy! !*h!:!Om! !2*A!	Jul 1, 1985 9:03AM Jul 18, 1985 11:13PM	26
Columnar day, date, and time of day	4	!Www! !Nnn! !dd!, !yyyy! !hh!:!Om! !2*A!	Mon Jul 1, 1985 9:03 AM Mon Jul 18, 1985 11:13 PM	30
Day, date, and time of day	5	!Www! !Nnn! !*d!, !yyyy! !*h!:!0m! !2*A!	Mon Jul 1, 1985 9:03 AM Mon Jul 18, 1985 11:13 PM	30
Columnar 24-hour time of day*	6	!0t!:!0m!	23:43 07:00	5
24-hour timeof day*	7	!*t!:!0m!	23:43 7:00	5
Columnar time of day	8	!hh!:!0m! !2*A!	11:43PM 7:00AM	8
Time of day	9	!*h!:!0m! !2*A!	11:43PM 7:00AM	8
Columnar numeric date	10	!0o!/!0d!/!0y!	11/18/85 01/01/01	8
Numeric date	11	!*o!/!*d!/!0y!	11/18/85 1/1/01	8
Time	12	!0h!:!0m!:!0s!	09:03:05	8
Temporary file name	13	!0t!:!0m!:!0s!.tmp 23:03:05.tmp	09:03:05.tmp	30
Long day and date	14	!W*w! !N*n! !*d!, !*y!	Monday June 3, 1985	45
Long date	15	!N*n! !*d!, !*y!	June 3, 1985	30
Columnar date/time for file names†	16	!0o!/!0d!/!0y!-!0h!:!0m!:!0s!!2*A!	01/01/01-07:09:05AM	40
Abbreviated date	17	!Nnn! !*d! !YYYY!	Jun 3, 1985	30

*For applications that want 24-hour time regardless of country.
†Various strings are normally appended to construct file names. No spaces should be allowed.

Figure B–1. NLS Templates (Page 1 of 1)

C

Message File Macro Definitions

To allow added flexibility in outputing messages, the messages in a message file can contain macros. Message file macros serve as place holders in a message for data that is to be inserted into the message at run time.

The macros may be expanded with data supplied by using message file operations such as ExpandLocalMsg, GetMsg, GetAltMsg, PrintMsg, or PrintAltMsg or by programs calling such operations.

Macros Not Under Program Control

Macro	Meaning
%U	Insert the workstation user name in the message. Note that %U expands to NULL if the user name in the Application System Control Block is of 0 length.
%D*n*.	Insert the system date/time in the message formatted with template number *n*.

See Appendix B for the date/time template formats, descriptions, and examples. The templates are defined in the NlsDateTables.Asm file.

%K*n*.	Insert the keycap text string for key cap index number *n* into the message.

See the description of GetNlsKeycapText in the *CTOS/Open Application Programming Interface Specification* for the key cap index numbers and their corresponding text strings. Keycap text is defined in the keycaptext.asm file.

Macros Under Program Control

Macro	Meaning
%*n*T	Insert the program-supplied parameter number n. T is a string denoting format options for the number *n*.

T can be any of the following values:

Value	Meaning
D*m*	date/time (where *m* is the template ID code)
N	unsigned decimal–based integer
H	unsigned hexadecimal–based integer (See format options below.)
S	string

If T is not specified, S is the default.

The macro %6D4, for example, means the sixth array element will be expanded as a date using template ID code 4 in the date/time templates.

See the descriptions of ExpandLocalMsg, GetMsg, GetAltMsg, PrintMsg and PrintAltMsg in the *CTOS/Open Application Programming Interface Specification*. *pRgSd* and *sRgSd* describe the array that you include in your program for supplying the parameters to be inserted into messages.

Format Options for Macro Option H

The hexadecimal macro option (H) expands the string to a fixed number of places based on the count of bytes (1, 2, or 4) in the string descriptor array element. As an example, a value of 100 can return one of three different strings, depending on the string size, as shown at the top of the next page.

Value	String Size (in bytes)	Returned String
100	1	64
100	2	0064
100	4	00000064

For other format options, use OutputWord or OutputQuad described in the *CTOS/Open Application Programming Interface Specification*.

D

Source Code Listings

This appendix contains the source code listings for each of the sample programs that appear in this manual. Not all the listings in the manual are part of sample programs. Some are independent code fragments.

In general, the listings are grouped based on the chapter in which they first appear. For example, if parts of a sample program appear both in Chapter 2, and in Chapter 7, the listing appears under the heading for Chapter 2. There is, however, a reference to that listing under the heading for Chapter 7.

The listings appear in the order in which they are used in a given chapter. If the first example in a chapter is extracted from Foo.c, then Foo.c appears first in the listings for that chapter. Foo.c does not reappear anywhere else in this appendix.

The sample programs in this appendix have been tested and should work on most hardware-software configurations, if they are compiled and linked correctly. However, they are not guaranteed to work in any configuration.

Chapter 1

Chapter 1 contains no listings.

Chapter 2

The program listings for Chapter 2 are listed in this section. Before each program is a paragraph which identifies the listings in which portions of it appear.

BSGeneric.c

This sample program is referenced by Listings 2–1, 2–2, 2–3, and 2–5.

```
/*************************************************************************/
/*                                                                       */
/* File Name: BsGeneric.c                                                */
/* Compiler:  Metaware C                                                 */
/* Date:      04/27/89                                                   */
/* Author:    A. Coleman/J. Crook                                        */
/*                                                                       */
/* This source file provides examples of byte stream operations.  The*/
/* program asks for a byte stream from the user. Once the byte stream*/
/* is opened, the user is asked to enter strings from the keyboard.  */
/* Each string is written to the byte stream with an echo of each    */
/* character to the video monitor. Entry of strings is completed when*/
/* the user enters a null string (by pressing the Return key).       */
/*                                                                   */
/* If the user specifies a disk byte stream (a file), it is open in  */
/* modify mode. Once the user completes the entry of the strings, the*/
/* file is reset and all line feeds are replaced by form feed        */
/* characters.                                                       */
/*                                                                   */
/* If the user specifies a video byte stream ([Vid]), it is open in  */
/* write mode.                                                       */
/*                                                                   */
/* If the user specifies an invalid byte stream, (e.g., [Kbd]), an   */
/* error is returned.                                                */
/*                                                                   */
/*************************************************************************/

/* file access constants */
#define WRITEMODE       0x6D77          /* 'mw' */
#define READMODE        0x6D72          /* 'mr' */
#define MODIFYMODE      0x6D6D          /* 'mm' */

/* allocation sizes */
#define STRSIZE         128
#define ALLOC_SIZE      0x100

/* input characters */
#define LINEFEED        0x0A
#define FORMFEED        0x0C

/* key definitions */
#define FINISH          0x04
#define CANCEL          0x07
#define RETURN          0x0A
#define GO              0x1B
```

```
/* error codes */
#define ERCOK           0x00
#define ERCEOF          0x01
#define ERCOPERATOR     0x04
#define ERCNOTIMPLEMENTED  0x07
#define ERCINVALIDBSWA 0x915
#define ERCBOGUS        0xFFFF

#define Syslit

/* define CTOS procedures to be used */
#define Beep
#define CheckErc
#define CheckPointBs
#define CloseByteStream
#define ErrorExit
#define GetBsLfa
#define NlsClass
#define OpenByteStream
#define ReadBSRecord
#define ReadByte
#define SetBsLfa
#define WriteBsRecord
#define WriteByte

#include <ctoslib.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

/* define the pre-opened byte streams */
extern   char     bsVid[];
extern   char     bsKbd[];

/* define my global variables */
char rgbString[STRSIZE];    /* general purpose string */
char MyBSWA[130];
int  cbString;
Word rgMyBuffer[512];

char rgbBsPrompt[] =
         "\nType a device or file spec for the Byte Stream you want."
         "\nPress CANCEL if you make a mistake:  ";
char rgbExit[] =
         "\n\nPress GO to confirm FINISH, or CANCEL to continue.";
char rgbInvalidBs[] =
                 "\n\nBytestream is invalid for output: ";
char rgbPrompt[] =
         "\n\nType a character string, then press RETURN:  ";


/*********************************************************************/
/*                                                                   */
/* HandleFinishKey                                                   */
/*                                                                   */
/* This procedure determines whether the user would like to          */
/* discontinue processing input strings.  It is called when the      */
/* user presses the FINISH key. A GO key causes the program to Exit, */
/* a Cancel returns to the point of invocation of this procedure.    */
/*                                                                   */
/*********************************************************************/
```

```
HandleFinishKey (char *BSWA)
   {
   char       bIn;
   ErcType    erc;
   Word       cbRet;

   CheckErc (WriteBsRecord (bsVid, rgbExit, strlen(rgbExit), &cbRet));
   erc = ERCOK;
   while (erc == ERCOK)
      {
      erc = ReadByte (BSWA, &bIn);
      switch (erc)
         {
      case ERCOK:      /* user pressed key other than FINISH, CANCEL */
      case ERCEOF:     /* user pressed FINISH */
         if (bIn == GO) ErrorExit (0);
         else Beep ();
         break;
      case ERCOPERATOR:  /* user pressed CANCEL key */
         return (-1);
      default:         /* error occurred on read */
         ErrorExit (erc);
         }
      }
   }


/*************************************************************************/
/*                                                                       */
/* GetString                                                             */
/*                                                                       */
/* This procedure returns an input string to the caller.  The string */
/* will either be the name of the byte stream or the name of an input*/
/* string being written to the byte stream previously opened.        */
/*                                                                       */
/*************************************************************************/

GetString (char *BSWA, char *rgb, Word cbMax)
   {
   char  bIn;
   char  class;
   char  i = 0;
   ErcTypeerc;

   /* zero the string */
   while ((rgb[i] != 0) && (i < cbMax))
      rgb[i++] = 0;

   /* collect the chars */
   i = 0;
   while (TRUE)
      {
      erc = ReadByte (BSWA, &bIn);
      if (erc == ERCEOF) HandleFinishKey (BSWA); /* FINISH key */

      else if (erc == ERCOPERATOR) return (-1);   /* CANCEL key */

      else
         {
         switch (bIn)

            {
```

```
            case GO:
            case RETURN:
               return (i);
            default:
             /* check if it's a letter, digit, or keyboard special char */
               CheckErc (NlsClass (NULL, bIn, &class));
               if (class <= 2)
                  {       /* valid character, echo it out */
                  CheckErc (WriteByte (bsVid, bIn));
                  rgb[i] = bIn;
                  if (i < STRSIZE) i++;
                  }
               else Beep (); /* invalid character entered, don't echo it*/
               break;

               } /* end of switch */
            } /* end of else */
         } /* end of while */
      return(0);
      }


/*************************************************************************/
/*                                                                       */
/* ReplaceLineFeeds                                                      */
/*                                                                       */
/* This procedure searches through the open byte stream replacing        */
/* each line feed with a form feed                                       */
/*                                                                       */
/*************************************************************************/

void ReplaceLineFeeds (char *BSWA)
   {
   char   bIn;
   ErcType erc;
   long   Lfa;

   /* flush my buffers */
   CheckErc (CheckPointBs (BSWA));
   erc = GetBsLfa (BSWA, &Lfa);

   if (erc == ERCOK)
      { /* it's a file */
      CheckErc (SetBsLfa (BSWA, 0));
      erc = ReadByte (BSWA, &bIn);
      while (erc == ERCOK)
         {
         if (bIn == LINEFEED)
            {  /* change this to a form feed just for fun */
            CheckErc (GetBsLfa (BSWA, &Lfa));
            CheckErc (SetBsLfa (BSWA, (Lfa - 1)));
            CheckErc (WriteByte (BSWA, FORMFEED));
            }
         /* get the next character */
         erc = ReadByte (BSWA, &bIn);
         }
      }
   else if (erc != ERCINVALIDBSWA) ErrorExit (erc);  /* eat this one */

   /* flush the buffer and close the file */
   CheckErc (CloseByteStream(BSWA));
   }
```

```
/*********************************************************************/
/*                                                                   */
/* GetByteStream                                                     */
/*                                                                   */
/* This procedure asks the user to enter the name of the desired byte*/
/* stream.  Once the user enters the string, it used in attempting to*/
/* open the corresponding byte stream in modify mode.  If an error   */
/* occurs, the same byte stream is opened in write mode.  If another */
/* error occurs, it is not a valid byte stream for output and the    */
/* user is told so.  The procedure then loops back for another name. */
/*                                                                   */
/*********************************************************************/

void GetByteStream (char *BSWA)
   {
   ErcType erc;
   Word   cbRet;

   do
      {
      CheckErc (WriteBsRecord (bsVid,rgbBsPrompt,sizeof(rgbBsPrompt),
                               &cbRet));
      cbString = GetString(bsKbd, rgbString, STRSIZE);
      if (cbString > 0)
         {
         erc = OpenByteStream (BSWA, rgbString, cbString, NULL, 0,
                               MODIFYMODE, rgMyBuffer, sizeof(rgMyBuffer));
         if (erc == ERCNOTIMPLEMENTED)
            {  /* Mode modify not applicable, check write */
            erc = OpenByteStream (BSWA, rgbString, cbString, NULL, 0,
                                  WRITEMODE, rgMyBuffer, sizeof(rgMyBuffer));
            if (erc == ERCNOTIMPLEMENTED)
               {  /* Mode write not applicable */
               CheckErc (WriteBsRecord (bsVid, rgbInvalidBs,
                                        strlen(rgbInvalidBs),&cbRet));
               CheckErc (WriteBsRecord (bsVid, rgbString, cbString,
                                 &cbRet));
               CheckErc (WriteByte (bsVid,LINEFEED));
               erc = ERCBOGUS;/* set bogus error so we can loop back */
               }
            }
         else CheckErc (erc);
         }
      else /* empty string, go get another one */
         {
         CheckErc (WriteByte (bsVid,LINEFEED));
         erc = ERCBOGUS; /* set bogus error so we can loop back */
         }
      }
   while (erc != 0); /* if they make a mistake, ask again */
   }


/*********************************************************************/
/*                                                                   */
/* main                                                              */
/*                                                                   */
/*********************************************************************/

main()
```

```
[
Word  cbRet;

GetByteStream (MyBSWA);

/* read input with echo, then output it  */
/* single RETURN stops the input request */
do
   [
   CheckErc (WriteBsRecord (bsVid, rgbPrompt, sizeof(rgbPrompt),
            &cbRet));
   cbString = GetString (bsKbd, rgbString, STRSIZE);
   if (cbString > 0)
      [
      CheckErc (WriteByte (MyBSWA, LINEFEED));
      CheckErc (WriteBsRecord (MyBSWA,rgbString,cbString, &cbRet));
      ]
   ]
while (cbString != 0);

ReplaceLineFeeds (MyBSWA);

return(0);
]
```

BSVidEscape.c

This sample program is referenced by Listing 2–4.

```
/*********************************************************************/
/*                                                                   */
/* File Name:BsVidEscape.c                                           */
/* Compiler:Metaware C                                               */
/* Date:04/29/89                                                     */
/* Author:A. Coleman/J. Crook                                        */
/*                                                                   */
/* This source file provides several examples of video byte stream   */
/* operations. The user is asked to enter video escape sequences from*/
/* the keyboard.  Each sequence starts with an introductory          */
/* character, 'E'.  All characters must be separated by a space, and */
/* the end of the escape sequence is indicated by a RETURN key.      */
/*                                                                   */
/* Two video frames are used in this program.  Frame 0 (the working  */
/* frame) is the same frame as the Executive uses for enabling user  */
/* input. This frame is used for managing the user entered video byte*/
/* streams.  Frame 1 (the status frame) is the same frame as the     */
/* Exeuctive title frame.  This frame is used for informing the user */
/* of the action to be taken.                                        */
/*                                                                   */
/*********************************************************************/

/* define constants */
#define WRITEMODE      0x6D77 /* 'mw' */
#define READMODE       0x6D72 /* 'mr' */
#define STRSIZE        128
#define LINEFEED       0x0A
#define ESCAPE         0xFF
#define ZERO           0
```

```
#define ONE             1

/* escape sequence characters */
#define COLSET          'C'
#define ERASE           'E'
#define FRAME           'F'
#define OFF             'F'
#define PAUSE           'P'
#define REDIRECT        'X'

/* key definitions */
#define FINISH          0x04
#define CANCEL          0x07
#define RETURN          0x0A
#define GO              0x1B

/* error codes */
#define ERCOK           0x00
#define ERCEOF          0x01
#define ERCOPERATOR     0x04
#define ERCINVALIDVIDESCAPE  0x2336

#define Syslit

/* define CTOS procedures to be used */
#define Beep
#define CheckErc
#define CloseByteStream
#define ErrorExit
#define NlsClass
#define OpenByteStream
#define QueryVidBs
#define ReadBSRecord
#define ReadByte
#define WriteBsRecord
#define WriteByte

#include <ctoslib.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

typedef struct {
        char    nFrame;
        char    cLines;
        char    cCols;
        char    nLine;
        char    nCol;
        char    fCurVisible;
        char    fPause;
        char    bAttr;
        char    fLiteral;
        char    rgbReserved[7];
        } VidStatusType;

/* define the pre-opened byte streams */
externcharbsVid[];
externcharbsKbd[];

/* define my global variables */
int     cbString;
char    rgbString[STRSIZE]; /* general purpose string */
```

```
char     rgbEscape[STRSIZE];
VidStatusTypestrVidStatus;

/* escape sequences */
char escToFrame0 [] ={ ESCAPE, REDIRECT, ZERO };
char escToFrame1 [] ={ ESCAPE, REDIRECT, ONE };
char escNoPause [] ={ ESCAPE, PAUSE, OFF };
char escInitCursor [] ={ ESCAPE, COLSET, ZERO, ZERO };
char escClearFrame [] ={ ESCAPE, ERASE, FRAME };

/* message arrays */
char rgbExit[] =
        "\nPress GO to confirm FINISH, or CANCEL to continue.";
char rgbIntro[] =
        "\nTo use this program, type 'E' followed by the escape "
        "sequence you want.";
char rgbInvEscape [] =
        "\nInvalid video escape sequence.";
char rgbPrompt[] =
        "\nType an escape sequence separated by spaces, then press "
        "RETURN: ";
char rgbTest[] =
        "*** Testing ***";

/*************************************************************************/
/*                                                                       */
/* ClearFrames                                                           */
/*                                                                       */
/* This procedure sets the cursor at the origin and clears the frame.*/
/*                                                                       */
/*************************************************************************/

ClearFrames ()
   {
   int   i;
   Word  cbRet;

   for (i = 0; i < 2; i++)
      {
      switch (i)
         {  /* set the frame number */
      case 0:
         CheckErc(WriteBsRecord(bsVid,escToFrame0, sizeof(escToFrame0),
                                &cbRet));
         break;
      case 1:
         CheckErc(WriteBsRecord(bsVid,escToFrame1, sizeof(escToFrame1),
                                &cbRet));
         /* turn off pause */
         CheckErc(WriteBsRecord (bsVid, escNoPause, sizeof(escNoPause),
                                 &cbRet));
         break;
         }

      /* set the cursor at the origin */
      CheckErc(WriteBsRecord(bsVid,escInitCursor,sizeof(escInitCursor),
                             &cbRet));
      /* clear the frame */
      CheckErc(WriteBsRecord(bsVid,escClearFrame,sizeof(escClearFrame),
                             &cbRet));
      }
```

```
   return (0);
   ]

/*********************************************************************/
/*                                                                   */
/* HandleFinishKey                                                   */
/*                                                                   */
/* This procedure determines whether the user wants to discontinue   */
/* processing input strings.  It is called when the user presses the */
/* Finish key.  A Go key causes the program to Exit, a Cancel returns*/
/* to the point of invocation of this procedure                      */
/*                                                                   */
/*********************************************************************/

HandleFinishKey (char *BSWA)
   {
   char       bIn;
   ErcType    erc;
   Word       cbRet;

   CheckErc (WriteBsRecord (bsVid, rgbExit, strlen(rgbExit), &cbRet));
   erc = ERCOK;
   while (erc == ERCOK)
      {
      erc = ReadByte (BSWA, &bIn);
      switch (erc)
         {
      case ERCOK:   /* user pressed key other than FINISH, CANCEL */
      case ERCEOF:  /* user pressed FINISH */
         if (bIn == GO) ErrorExit (0);
         else CheckErc (Beep ());
         break;
      case ERCOPERATOR:  /* user pressed CANCEL key */
         return (-1);
      default: /* error occurred on read */
         ErrorExit (erc);
         }
      }
   ]

/*********************************************************************/
/*                                                                   */
/* GetString                                                         */
/*                                                                   */
/* This procedure returns an input string to the caller.  The string */
/* will either be the name of the byte stream or the name of an input*/
/* string being written to the byte stream previously opened.        */
/*                                                                   */
/*********************************************************************/

GetString(char *BSWA, char *rgb, Word cbMax)
   {
   char       bIn;
   char       class;
   char       i = 0;
   ErcType    erc;

   /* zero the string */
   while ((rgb[i] != 0) && (i < cbMax))
      rgb[i++] = 0;

   /* collect the chars */
```

```
   i = 0;
   while (TRUE)
      {
      erc = ReadByte (BSWA, &bIn);
      if (erc == ERCEOF) HandleFinishKey (BSWA); /* Finish key */

      else if (erc == ERCOPERATOR) return(-1);   /* Cancel key */

      else
         {
         switch (bIn)
            {
         case GO:
         case RETURN:
            return (i);
         default:
            /* check for a letter, digit or keyboard special char */
            CheckErc (NlsClass (NULL, bIn, &class));
            if (class <= 2)
               {  /* valid character, echo it out */
               CheckErc (WriteByte (bsVid, bIn));
               rgb[i] = bIn;
               if (i < STRSIZE) i++;
               }
            else Beep ();  /* invalid char entered, don't echo it */
            break;

            } /* end of switch */
         } /* end of else */
      } /* end of while */
   return(0);
   }

/*********************************************************************/
/*                                                                   */
/* PrintEscape                                                       */
/*                                                                   */
/* This procedure reads the video status buffer from QueryVidBs to   */
/* get the current cursor position, saves it, performs the specified */
/* byte stream operation, and restores the cursor position.          */
/*                                                                   */
/*********************************************************************/

PrintEscape (char *rgbEscape, char cbEscape)
   {
   char       i;
   char       nCol, nFrame, nLine;
   ErcType    erc;
   Word       cbRet;

   /* save the current cursor location */
   CheckErc (QueryVidBs (bsVid, &strVidStatus));
   nFrame = strVidStatus.nFrame;
   nCol   = strVidStatus.nCol;
   nLine  = strVidStatus.nLine;

   /* change to frame 0 (working frame) */
   CheckErc (WriteBsRecord (bsVid, escToFrame0, sizeof(escToFrame0),
                            &cbRet));

   /* now print out the user entered escape sequence */
   CheckErc (WriteByte (bsVid, ESCAPE));
```

```
    for (i = 0; i < cbEscape; i++)
       {
       erc = WriteByte (bsVid, *(rgbEscape+i));
       if (erc == ERCINVALIDVIDESCAPE)
          {
          /* change to frame 1 (title frame) */
          CheckErc(WriteBsRecord(bsVid,escToFramel,sizeof(escToFramel),
                                 &cbRet));
          /* print out error message */
          CheckErc (WriteBsRecord (bsVid, rgbInvEscape,
                                 strlen(rgbInvEscape), &cbRet));
          return (-1);
          }
       }

    /* output the sample string */
    CheckErc (WriteBsRecord (bsVid, rgbTest, strlen(rgbTest), &cbRet));

    /* reset the frame just in case, can't use a canned sequence */
    rgbEscape[0] = ESCAPE;
    rgbEscape[1] = REDIRECT;
    rgbEscape[2] = nFrame;
    CheckErc (WriteBsRecord (bsVid, rgbEscape, 3, &cbRet));

    /* reset the cursor postion */
    rgbEscape[1] = COLSET;
    rgbEscape[2] = nCol;
    rgbEscape[3] = nLine;
    CheckErc (WriteBsRecord (bsVid, rgbEscape, 4, &cbRet));
    return (0);
    }

/*************************************************************************/
/*                                                                       */
/* main                                                                  */
/*                                                                       */
/*************************************************************************/

main()
    {
    char  i, j;
    char  *pString;
    Word  cbRet;

    ClearFrames ();
    CheckErc (WriteBsRecord(bsVid, rgbIntro, sizeof(rgbIntro), &cbRet));

    /* get escape sequence, do it, ask for a new one */
    do
       {
       i = 0;
       j = 0;
       CheckErc (WriteBsRecord (bsVid, rgbPrompt, sizeof(rgbPrompt),
                                &cbRet));
       cbString = GetString(bsKbd, rgbString, STRSIZE);

       if (cbString > 0)
           if (rgbString[i] == 'E')
             { /* we have the start of an escape sequence */
             i++;
             while (rgbString[i] != 0)
                {
```

```
            if (isalpha (rgbString[i]) != FALSE)
               {
               rgbEscape[j] = rgbString[i];
               i++;
               j++;
               }
            else if (isdigit (rgbString[i]) != FALSE)
               {
               pString = &(rgbString[i]);
               rgbEscape[j] = atoi (pString);
               j++;
               while (isdigit (rgbString[i]) != FALSE) i++;
               }
            else i++;
            } /* end while rgbString[i] */

         rgbEscape[j] = 0; /* null-terminate it just for fun */
         PrintEscape(rgbEscape, j);
         } /* end if it starts with E */
      else CheckErc (Beep ());
   } /* end if strsize > 0 */
while (cbString != 0);

return(0);
}
```

FileCalls.c

This sample program is referenced by Listings 2–6 and 2–7.

```
/*************************************************************************/
/*                                                                       */
/* File Name:FileCalls.c                                                 */
/* Compiler:Metaware C                                                   */
/* Date:05/2/89                                                          */
/* Author:A. Coleman                                                     */
/*                                                                       */
/* This source file provides examples of direct file operations.  The*/
/* program asks for a file name from the user. Once the file          */
/* is opened, the user is asked to enter strings from the keyboard.  */
/* Each string is written to the file along with an echo of each     */
/* character to the video monitor. Entry of strings is completed when*/
/* the user enters a null string (by pressing the Return key).       */
/*                                                                   */
/* Once the user completes the entry of the strings, all line feed   */
/* characters in the file are replaced by form feed characters.      */
/*                                                                   */
/*************************************************************************/

/* file access constants */
#define WRITEMODE      0x6D77 /* 'mw' */
#define READMODE       0x6D72 /* 'mr' */
#define MODIFYMODE     0x6D6D /* 'mm' */

/* allocation sizes */
#define STRSIZE        256
#define ALLOC_SIZE     0x100
#define SMYBUFFER      512
```

```
#define MYBUFFERMAX    511

/* input characters */
#define LINEFEED       0x0A
#define FORMFEED       0x0C

/* key definitions */
#define FINISH         0x04
#define CANCEL         0x07
#define RETURN         0x0A
#define GO             0x1B

/* error codes */
#define ERCOK          0x00
#define ERCEOF         0x01
#define ERCOPERATOR    0x04
#define ERCNOTIMPLEMENTED  0x07
#define ERCNOSUCHFILE      0x0cb /* 203 decimal */
#define ERCINVALIDBSWA     0x915
#define ERCBOGUS       0xFFFF

#define Syslit

/* define CTOS procedures to be used */
#define Beep
#define ChangeFileLength
#define CheckErc
#define CheckPointBs
#define CloseByteStream
#define CloseFile
#define CreateFile
#define ErrorExit
#define GetBsLfa
#define GetFileStatus
#define NlsClass
#define OpenByteStream
#define OpenFile
#define Read
#define ReadBSRecord
#define ReadByte
#define SetBsLfa
#define SetFileStatus
#define Write
#define WriteBsRecord
#define WriteByte

#include <ctoslib.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

/* define the pre-opened byte streams */
extern   char     bsVid[];
extern   char     bsKbd[];

/* define my global variables */
char rgbString[STRSIZE];/* general purpose string */
int  cbString;
char rgbMyBuffer[SMYBUFFER];

char rgbBsPrompt[] =
        "\nType a file spec for the file you want."
```

```
        "\nPress CANCEL if you make a mistake:  ";
char rgbExit[] =
        "\n\nPress GO to confirm FINISH, or CANCEL to continue.";
char rgbInvalidBs[] =
              "\n\nFilespec is invalid. ";
char rgbPrompt[] =
        "\n\nType a character string, then press RETURN:  ";
char    rgbCancel[] =
        "\n\nPress any key to continue, or press CANCEL to exit.";

/*************************************************************************/
/*                                                                       */
/* HandleFinishKey                                                       */
/*                                                                       */
/* This procedure determines whether the user would like to quit     */
/* processing input strings.  It is called when the user presses the */
/* FINISH key.  A GO key causes the program to Exit, a Cancel returns*/
/* to the point of invocation of this procedure.                         */
/*                                                                       */
/*************************************************************************/

HandleFinishKey (char *BSWA)
   {
   char  bIn;
   ErcTypeerc;
   Word  cbRet;

   CheckErc (WriteBsRecord (bsVid, rgbExit, strlen(rgbExit), &cbRet));
   erc = ERCOK;
   while (erc == ERCOK)
      {
      erc = ReadByte (BSWA, &bIn);
      switch (erc)
         {
      case ERCOK:   /* user pressed key other than FINISH, CANCEL */
      case ERCEOF:  /* user pressed FINISH */
         if (bIn == GO) ErrorExit (0);
         else Beep ();
         break;
      case ERCOPERATOR:  /* user pressed CANCEL key */
         return (-1);
      default: /* error occurred on read */
         ErrorExit (erc);
         }
      }
   }

/*************************************************************************/
/*                                                                       */
/* GetString                                                             */
/*                                                                       */
/* This procedure returns an input string to the caller.  The string */
/* will either be the name of the byte stream or the name of an input*/
/* string being written to the byte stream previously opened.        */
/*                                                                       */
/*************************************************************************/

GetString (char *BSWA, char *rgb, Word cbMax)
   {
   char      bIn;
   char      class;
   char      i = 0;
```

```
    ErcType    erc;

    /* zero the string */
    while ((rgb[i] != 0) && (i < cbMax))
       rgb[i++] = 0;

    /* collect the chars */
    i = 0;
    while (TRUE)
       {
       erc = ReadByte (BSWA, &bIn);
       if (erc == ERCEOF) HandleFinishKey (BSWA);  /* FINISH key */

       else if (erc == ERCOPERATOR) return (-1);   /* CANCEL key */

       else
          {
          switch (bIn)

             {
          case GO:
          case RETURN:
             return (i);
          default:
             /* check for a letter, digit, or keyboard special char */
             CheckErc (NlsClass (NULL, bIn, &class));
             if (class <= 2)
                {  /* valid character, echo it out */
                CheckErc (WriteByte (bsVid, bIn));
                rgb[i] = bIn;
                if (i < STRSIZE) i++;
                }
             else Beep ();  /* invalid char entered, don't echo it */
             break;

             } /* end of switch */
          } /* end of else */
       } /* end of while */
    return(0);
    }

/*********************************************************************/
/*                                                                   */
/* GetFile                                                           */
/*                                                                   */
/* This procedure asks the user to enter the name of the desired file*/
/* Once the user enters the string, it used in attempting to         */
/* open the corresponding file in modify mode.                       */
/* If the user enters a null string, the procedure asks again.       */
/*                                                                   */
/*********************************************************************/

Word GetFile ()
    {
    ErcType erc;
    Word  fh = 0;
    Word  cbRet;
    DWord qZero = 0;

    do
       {
       CheckErc (WriteBsRecord (bsVid,rgbBsPrompt,sizeof(rgbBsPrompt),
```

```
                                &cbRet));
        cbString = GetString(bsKbd, rgbString, STRSIZE);
        if (cbString > 0)
           {
           erc=OpenFile(&fh,rgbString,strlen(rgbString),NULL,0,
                        MODIFYMODE);
           if(erc == ERCNOSUCHFILE) {
              /* create the file if it doesn't exist */
              CheckErc(CreateFile(rgbString,strlen(rgbString),NULL,0,0));
              CheckErc(OpenFile(&fh,rgbString,strlen(rgbString),NULL,0,
                                MODIFYMODE) );
              }
           else {
              CheckErc(erc);
              /* reset file size and EOF pointer */
              CheckErc( SetFileStatus(fh, 6, &qZero, sizeof(qZero)) );
              CheckErc( ChangeFileLength(fh, qZero) );
              }
           }
        else /* empty string, go get another one */
           {
           CheckErc (WriteByte (bsVid,LINEFEED));
           erc = ERCBOGUS; /* set bogus error so we can loop back */
           }
        }
     while (erc != 0); /* if they make a mistake, ask again */
     return(fh);
     }

/*********************************************************************/
/*  Store the data                                                   */
/* Receives a file handle and a count of bytes.  Stores that many    */
/* bytes at the end of the file associated with the handle.  Adds    */
/* to the file size to accomodate the new data, and resets the       */
/* EOF pointer.                                                      */
/*********************************************************************/

void StoreData(Word fhMyHandle, Word cBytes) {
   DWord qEOFPtr;
   DWord lfa;
   Word  sDataRet;
   /* get current lfa and EOF pointer */
   CheckErc( GetFileStatus(fhMyHandle, 0, &lfa, sizeof(lfa)) );
   CheckErc( GetFileStatus(fhMyHandle, 6, &qEOFPtr, sizeof(qEOFPtr)) );
   /* increase file size */
   lfa += 512;
   CheckErc( ChangeFileLength(fhMyHandle, lfa ) );
   CheckErc(Write(fhMyHandle,rgbMyBuffer,SMYBUFFER,qEOFPtr,
                  &sDataRet) );
   /* reset EOF pointer */
   qEOFPtr += cBytes;
   CheckErc(SetFileStatus(fhMyHandle, 6, &qEOFPtr, sizeof(qEOFPtr) ) );
}


/*********************************************************************/
/*                                                                   */
/* main                                                              */
/*                                                                   */
/*********************************************************************/

main()
```

```
   {
   ErcType    erc;
   Word       cbRet;
   Word       fhMyHandle;
   Word       iCharCount = 0;
   Word       iTemp = 0;
   Byte       bMyChar;

   fhMyHandle = GetFile ();

do {
   /* read input with echo, then output it  */
   /* single RETURN stops the input request */
   do
      {
      CheckErc(WriteBsRecord(bsVid,rgbPrompt,sizeof(rgbPrompt),
                             &cbRet));
      cbString = GetString (bsKbd, rgbString, STRSIZE);
      if (cbString > 0)
         {
         rgbMyBuffer[iCharCount] = LINEFEED;
         iCharCount++;
         for(iTemp=0;iTemp<cbString;iTemp++) {
            if(iCharCount == SMYBUFFER) {
               StoreData(fhMyHandle, iCharCount);
               iCharCount = 0;
               }
            rgbMyBuffer[iCharCount] = rgbString[iTemp];
            iCharCount++;
            }
         }
      }
   while (cbString != 0);

   CheckErc(WriteBsRecord(bsVid,rgbCancel,sizeof(rgbCancel),&cbRet));
   erc = ReadByte(bsKbd, &bMyChar);
   if(erc == ERCOPERATOR)
      break;
   }
   while(TRUE);

   StoreData(fhMyHandle, iCharCount);
   CheckErc( CloseFile(fhMyHandle) );

   return(0);
   }
```

WildCard.c

This sample program is referenced by Listing 2–8.

```
/*************************************************************************/
/*                                                                       */
/* File Name:WildCard.c                                                  */
/* Compiler:Metaware C                                                   */
/* Date:04/30/89                                                         */
/* Author:A. Coleman/J. Crook                                            */
/*                                                                       */
```

```
/* This program asks for a file specification from the user.  If the */
/* file specification contains wild card characters, they are        */
/* expanded and the resulting files are displayed to the video.      */
/*                                                                   */
/*********************************************************************/

/* define CTOS calls */
#define Beep
#defineCheckErc
#defineErrorExit
#defineExit
#defineNlsClass
#defineReadKbd
#defineWildCardInit
#defineWildCardNext
#defineWriteBsRecord
#defineWriteByte
#defineSysLit

#define STRSIZE   80
#define MEMSIZE   800
#define FINISH    0x04
#define CANCEL    0x07
#define RETURN    0x0A
#define GO        0x1B

#define ERCOK     0
#define ERCEOF    0x01
#define ERCOPERATOR    0x07
#define ERCINSUFFMEM   0x190

#include<ctoslib.h>
#include<stdlib.h>
#include<string.h>

typedef struct { Pointer pb; Word cb; } sdType;

extern char *bsVid;

char rgbExit[] =
            "\nPress GO to confirm FINISH, or CANCEL to continue.";
char rgbPrompt[] =
            "\nType a file specification, then press RETURN:  ";

/*********************************************************************/
/*                                                                   */
/* GetString                                                         */
/*                                                                   */
/* This procedure returns an input string containing the file spec   */
/* entered by the user.                                              */
/*                                                                   */
/*********************************************************************/

GetString(char *rgb)
   {
   char  bIn;
   char  class;
   char  i = 0;
   Word  cbRet;

   /* zero the string */
   while (rgb[i] != 0)
```

```
      rgb[i++] = 0;

   /* collect the chars */
   i=0;
   while(TRUE)
      {
      CheckErc (ReadKbd (&bIn));
      switch (bIn)
         {
      case CANCEL:
         return (-1);
      case FINISH:
         CheckErc (WriteBsRecord (&bsVid, rgbExit, strlen(rgbExit),
                   &cbRet));
         CheckErc (ReadKbd (&bIn));
         if (bIn == GO) ErrorExit (0);
         else if (bIn != CANCEL) Beep ();
         return (-1);
      case GO:
      case RETURN:
         CheckErc (WriteByte (&bsVid, RETURN));
         return (i);
      default:
         CheckErc (NlsClass (NULL, bIn, &class) );
         /* check if it's a letter, digit, or keyboard special char */
         if(class <= 2)
            {
            CheckErc (WriteByte (&bsVid, bIn));
            rgb[i] = bIn;
            if (i < STRSIZE) i++;
            }
         else Beep ();  /* invalid character entered, don't echo it */
         break;

         } /* end of switch */
      } /* end of while */
   return (0);
   }

/*************************************************************************/
/*                                                                       */
/* ListMatches                                                           */
/*                                                                       */
/* This procedure performs the expansion of the file specification       */
/* and lists all files matching the input expression.                    */
/*                                                                       */
/*************************************************************************/

ErcType ListMatches (char *pFs)
   {
   ErcTypeerc = 0;
   sdTypesd Fs;
   Word  *pBuffer;
   Word  cbRet;

   sdFs.pb = NULL;
   sdFs.cb = 0;

   if ((pBuffer = malloc (MEMSIZE)) == NULL) ErrorExit (ERCINSUFFMEM);

   /* initialize the wild card expansion environment */
   CheckErc (WildCardInit (pFs, strlen(pFs), pBuffer, MEMSIZE));
```

```
    while (erc == ERCOK)
       [
       /* get the next file matching the wild carded specification */
       erc = WildCardNext (pBuffer, &sdFs);
       if (erc == ERCOK)
          [
          CheckErc (WriteBsRecord (&bsVid, sdFs.pb, sdFs.cb, &cbRet));
          CheckErc (WriteByte (&bsVid, RETURN));
          ]
       else if (erc == ERCEOF) return(0);
       else return (erc);
       ]
    ]

/*************************************************************************/
/*                                                                       */
/* main                                                                  */
/*                                                                       */
/*************************************************************************/

main()
{
    char  rgbString[STRSIZE];
    Word  cbRet, cbString;

    while (TRUE)
       [
       CheckErc (WriteBsRecord (&bsVid, rgbPrompt, strlen(rgbPrompt),
                                &cbRet));
       cbString = GetString (rgbString);
       if (cbString != 0)
          CheckErc (ListMatches (rgbString));
       else break;
       ] /* end of while */
    return(0);
    ]
```

VAMText.c

This sample program is referenced by Listings 2–9 and 2–10.

```
/*************************************************************************/
/*                                                                       */
/* File Name:VAMText.C                                                   */
/* Compiler:Metaware C                                                   */
/* Date:05/04/89                                                         */
/* Author:A. Coleman/J. Crook                                            */
/*                                                                       */
/* This program asks for a list of strings from the user. Each string*/
/* is added to a linked list. When the user enters a null string, the*/
/* list is printed out in reverse order.                                 */
/*                                                                       */
/* The main areas to look at in this are the InitVideo routine which */
/* initializes the video environment and the GetString routine which */
/* uses the  TextEdit routine for processing user input.                 */
/*                                                                       */
/*************************************************************************/
```

```
/* define constants */
#define STRSIZE20
#define MAXBUF560 /* enough buffer for 7 lines */

/* Key definitions */
#define HELP       0x00
#define FINISH     0x04
#define CANCEL     0x07
#define RETURN     0x0A
#define F1         0x15
#define F2         0x16
#define F3         0x17
#define F4         0x18
#define F5         0x19
#define F6         0x1A
#define GO         0x1B
#define F7         0x1C
#define F8         0x1D
#define F9         0x1E
#define F10        0x1F
#define ESCAPE     0xFF

/* Screen characters */
#define VERTICALBAR        0x16
#define THINBORDERCHAR     0xDA
#define THICKBORDERCHAR    0xCE

/* Frame definitions */
#define MAINFRAME      0
#define MSGFRAME       1
#define TITLEFRAME     2
#define FKEYFRAME      3
#define HELPFRAME      4

/* Character-Screen attributes */
#define REFRESH        1
#define HALFBRITEREVERSE  5

/* define CTOS procedures to be used */
#define Syslit

#define Beep
#define CheckErc
#define Delay
#define ErrorExit
#define InitCharMap
#define InitVidFrame
#define PosFrameCursor
#define PutFrameChars
#define PutFrameCharsAndAttrs
#define PutFrameAttrs
#define QueryFrameBounds
#define QueryFrameCharsAndAttrs
#define QueryFrameCursor
#define QueryVidHdw
#define ReadKbd
#define ResetFrame
#define ResetVideo
#define ScrollFrame
#define SetScreenVidAttr
#define TextEdit
```

```
#include <ctoslib.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/* TextEdit buffer type */
typedef struct
     { char _far*prgch;
       Word   cchMax;
       Word   cchMac;
       Word   ichCursor;
     } TdbType;

/* a linked list to store stuff in */
typedef struct LListType *LListPtr;

typedef struct LListType
     { LListPtrnext;
       char   str[STRSIZE];
     } LListType;

char      str[STRSIZE+1];/* general purpose string */
Word      iCurCol, iCurLine;/* cursor position in frame 0 */
Word      nCols, nLines;  /* lines/cols on screen */
Word      sFrame0; /* lines in frame 0 */
int       strsize;

LListPtrllElement;
LListPtrllRoot;
TdbType  textBlock;
TdbType  *tbPtr;

/* frame buffers for Help */
Byte pbTextBuf[MAXBUF];
Byte pbAttrBuf[MAXBUF];

/* arrays */
char rgbOut[80];
char rgbTitle[] =
          "VAM and VDM Initialization Program";
char rgbFkeys[] =
          "  F1       F2       F3       F4       F5     "
          "  F6       F7       F8       F9       F10    ";
char rgbFuncKeyMsg[] =
          "You pressed a function key.  Press any key to continue.";
char rgbExitString[] =
          "Press GO to confirm FINISH, or CANCEL to continue. ";
char rgbIntro[] =
          "An empty string on input will display all the strings "
          "entered to that point.";
char rgbPrompt[] =
          "Type some characters, then press RETURN:  ";
char rgbContinueMsg[] =
          "Press GO to display output, or press CANCEL "
          "to enter more input.";
char rgbHelpMsg[] =
          "You Pressed the HELP key.  Press CANCEL to continue.";

/* forward procedure declarations */
void PrintMsg (char *pMessage);
```

```
/*****************************************************************/
/*                                                               */
/* DisplayHelp                                                   */
/*                                                               */
/* This procedure displays a help message, when the user responds to */
/* the message, the original screen is restored.                 */
/*                                                               */
/*****************************************************************/

void DisplayHelp()
   {
   char  bIn;
   Word sFrameBuf;

   sFrameBuf = nCols * 3;
   CheckErc (QueryFrameCharsAndAttrs (HELPFRAME, 0, 0, pbTextBuf,
                                      sFrameBuf,pbAttrBuf,sFrameBuf));
   CheckErc (ResetFrame(HELPFRAME));

   /* display the frame border and message */
   memset (&rgbOut,  THINBORDERCHAR, nCols);
   CheckErc (PutFrameChars (HELPFRAME, 0, 0, rgbOut, nCols) );
   CheckErc (PutFrameChars (HELPFRAME, 0, 2, rgbHelpMsg,
                            sizeof(rgbHelpMsg)));

   CheckErc (ReadKbd (&bIn));
   while (bIn != CANCEL)
      {
      CheckErc (Beep ());
      CheckErc (ReadKbd (&bIn));
      }

   CheckErc (PutFrameCharsAndAttrs (HELPFRAME, 0, 0, pbTextBuf,
                                    sFrameBuf, pbAttrBuf, sFrameBuf) );
   }

/*****************************************************************/
/*                                                               */
/* DisplaySoftKeys                                               */
/*                                                               */
/* This procedure displays the function key menu                 */
/*                                                               */
/*****************************************************************/

void DisplaySoftKeys ()
   {
   Word  cbOut, i;
   structFunctionKeyType
            {
            Byte FKeyStart;
            Byte FKeyCount;
            } rgCol[] = { {0,23}, {24,31}, {56,23} };


   /* construct the function key display */
   for (i = 0; i < 80; i++)
      switch (i)
         {
      case  7:
      case 15:
      case 31:
      case 39:
```

```
            case 47:
            case 63:
            case 71:
               rgbOut[i] = VERTICALBAR;
               break;
            default:
               rgbOut[i] = rgbFkeys[i];
               }

         /* display the function keys */
         cbOut = sizeof (rgbOut);
         CheckErc (PutFrameChars (FKEYFRAME, 0, 2, rgbOut, cbOut));

         /* display the frame border */
         memset (&rgbOut, THICKBORDERCHAR, cbOut);
         CheckErc (PutFrameChars (FKEYFRAME, 0, 0, rgbOut, cbOut));


         for (i = 0; i <= 2; i++)
            CheckErc (PutFrameAttrs (FKEYFRAME, rgCol[i].FKeyStart, 2,
                                     HALFBRITEREVERSE, rgCol[i].FKeyCount));

         }

/*************************************************************************/
/*                                                                       */
/* GetString                                                             */
/*                                                                       */
/* This procedure reads a string from the keyboard and echoes it to      */
/* one of the video frames.                                              */
/*                                                                       */
/*************************************************************************/

int GetString (char *str, int iFrame)
   {
   char  bIn; /* the character */
   char  i = 0;
   ErcTypeerc;

   /* set up buffer for TextEdit */
   tbPtr = &textBlock;
   textBlock.prgch= str;
   textBlock.cchMax= STRSIZE;
   textBlock.cchMac= 0;
   textBlock.ichCursor= 0;

   /* initialize string */
   while ((str[i] != 0) && (i < STRSIZE))
     str[i++] = 0;

   /* make cursor visible */
   CheckErc (PosFrameCursor (iFrame, iCurCol, iCurLine));

   /* get the chars */
   while (TRUE)
      {
      CheckErc (ReadKbd (&bIn));
      switch (bIn)
         {
      case FINISH:
         PrintMsg (rgbExitString);
         CheckErc (ReadKbd (&bIn));
```

```
            if (bIn == GO)ErrorExit(0);
            else PrintMsg (NULL);
            break;
         case CANCEL:
            return(-1);
         case GO:
         case RETURN:
            /* make cursor invisible */
            CheckErc (PosFrameCursor (iFrame, ESCAPE, ESCAPE));
            return (textBlock.cchMac);
         case HELP:
            /* turn off cursor */
            CheckErc (PosFrameCursor(iFrame, ESCAPE, ESCAPE));
            DisplayHelp ();
            /* turn it back on */
           CheckErc(PosFrameCursor(iFrame,(iCurCol+textBlock.ichCursor),
                                   iCurLine));
            break;
         case F1:
         case F2:
         case F3:
         case F4:
         case F5:
         case F6:
         case F7:
         case F8:
         case F9:
         case F10:
            /* tell them they pressed a function key */
            PrintMsg (rgbFuncKeyMsg);
            CheckErc (ReadKbd (&bIn) );
            PrintMsg (NULL);
            break;
         default:
            /* add the char to the string if it's a valid char */
            erc = TextEdit (bIn, tbPtr);
            /* beep if it isn't */
            if (erc != 0)
               {
               CheckErc (Beep ());
               break;
               }
            /* display modified TextEdit buffer, and move the cursor */
            CheckErc (PutFrameChars (iFrame, iCurCol, iCurLine,
                                     textBlock.prgch, textBlock.cchMax));
            CheckErc (PosFrameCursor(iFrame,(iCurCol+textBlock.ichCursor),
                                     iCurLine));
            break;
            } /* end of switch */
         } /* end of while */
      return(0);
      }

/*********************************************************************/
/*                                                                   */
/* InitVideo                                                         */
/*                                                                   */
/* This procedure initializes the video display frames: the main     */
/* working frame, the title frame, the message frame and the function*/
/* key frame.                                                        */
/*                                                                   */
/*********************************************************************/
```

```
void InitVideo ()
   [
   Word  cb, sMap; /* counter, char map */
   Byte  rgbVidHdw[3];

   /* find out the screen size */
   CheckErc (QueryVidHdw (&rgbVidHdw, sizeof(rgbVidHdw)));
   nLines = rgbVidHdw[1];
   nCols  = rgbVidHdw[2];

   /* reset the screen */
   CheckErc (ResetVideo (nCols, nLines, TRUE, 0x20, &sMap));

   /* set the size of frame 0 */
   sFrame0 = nLines - 6;

   /* Initialize the frames */
   CheckErc (InitVidFrame (MAINFRAME,0,3,nCols,sFrame0,0,
                           THINBORDERCHAR,0,0,0));
   CheckErc (InitVidFrame (TITLEFRAME,0,0,nCols,3,0,
                           THICKBORDERCHAR,0,0,0));
   CheckErc (InitVidFrame (MSGFRAME,0,nLines-2,nCols,1,0,0,0,0,0) );
   CheckErc (InitVidFrame (FKEYFRAME,0,nLines-3,nCols,3,0,0,0,0,0) );
   CheckErc (InitVidFrame (HELPFRAME,0,nLines-6 ,nCols, 3,0,0,0,0,0));

   CheckErc (InitCharMap (0, sMap));
   CheckErc (SetScreenVidAttr (REFRESH, TRUE));

   /* reset the frames */
   CheckErc (ResetFrame (MAINFRAME));
   CheckErc (ResetFrame (TITLEFRAME));
   CheckErc (ResetFrame (MSGFRAME));
   CheckErc (ResetFrame (FKEYFRAME));

   /* display the title */
   cb = sizeof (rgbTitle);
   CheckErc (PutFrameChars(TITLEFRAME,((nCols-cb)/2),0,rgbTitle,cb));

   /* display the frame border */
   cb = sizeof (rgbOut);
   memset (&rgbOut,  THINBORDERCHAR, cb);
   CheckErc (PutFrameChars (TITLEFRAME, 0, 2, rgbOut, cb));

   /* display the function keys */
   DisplaySoftKeys();
   ] /*   end InitVideo*/

/*********************************************************************/
/*                                                                   */
/* PrintMsg                                                          */
/*                                                                   */
/* This procedure displays a message in the message frame.  If the   */
/* message pointer is null, the message frame is cleared.            */
/*                                                                   */
/*********************************************************************/

void PrintMsg (char *pMessage)
   [
   if (pMessage == NULL)
      CheckErc (ResetFrame(MSGFRAME) );
   else
```

```
      CheckErc(PutFrameChars(MSGFRAME,0,0,pMessage,strlen(pMessage)));
   }

/*********************************************************************/
/*                                                                   */
/* PrintLineWithScroll                                               */
/*                                                                   */
/* This procedure displays a line of text and scrolls the screen if  */
/* necessary.                                                        */
/*                                                                   */
/*********************************************************************/

void PrintLineWithScroll (Word iFrame, Word *iCol, Word *iLine,
                          char *pMsg, Word cbMsg)
   {
   WordnCols, nLines;

   /* find out the maximum lines and cols */
   CheckErc (QueryFrameBounds (iFrame, &nCols, &nLines));
   nCols--; nLines--;

   /* display the message */
   CheckErc (PutFrameChars (iFrame, *iCol, *iLine, pMsg, cbMsg));

   /* increment line counter if we're not on last line, else scroll */
   if (*iLine < nLines) (*iLine)++;
   else CheckErc (ScrollFrame (iFrame, 0, ESCAPE, 1, TRUE));
   }

/*********************************************************************/
/*  main                                                             */
/*********************************************************************/
main()
   {
   Word  cbPrompt;
   char  bIn;

   llRoot = NULL;
   InitVideo();

   /* display the introduction */
   CheckErc (PutFrameChars (MAINFRAME, iCurCol, iCurLine++,
                            rgbIntro, sizeof(rgbIntro)));

   cbPrompt = strlen (rgbPrompt);
   while (TRUE)
      {

      /* display the prompt */
      CheckErc (PutFrameChars (MAINFRAME, iCurCol, iCurLine,
                               rgbPrompt, cbPrompt));

      /* put the cursor after the prompt, then get the input string */
      iCurCol = cbPrompt;
      strsize = GetString (str, MAINFRAME);

      /* reset cursor and scroll if necessary */
      iCurCol = 0;
      if (iCurLine < (sFrame0 - 1)) iCurLine++;
      else CheckErc (ScrollFrame (MAINFRAME, 0, 0xFF, 1, TRUE));

      /* if the string is not empty */
```

```
   if (strsize > 0)
      { /* put it in the linked list */
      llElement = malloc (sizeof (LListType));
      strcpy (llElement->str, str);
      llElement->next = llRoot;
      llRoot = llElement;
      }
   else
      {
      PrintMsg (rgbContinueMsg);
   while (TRUE)
         {
         CheckErc (ReadKbd (&bIn));
         if ((bIn == GO) || (bIn == CANCEL))
            {
            PrintMsg (NULL);
            break;
            }
         else CheckErc (Beep ());
         }
      if (bIn == GO) break;
      }
   } /* end of while */

llElement = llRoot;

/* print the strings we got */
while (llElement != NULL)
   {
   PrintLineWithScroll(MAINFRAME,&iCurCol,&iCurLine,llElement->str,
                       strlen (llElement->str));
   llElement = llElement->next;
   }

CheckErc (Delay (30));
return(0);
}
```

Chapter 3

Chapter 3 contains the following listings.

Erc80Handler.c

This sample program is referenced by Listings 3–1 and 3–2.

```
/*************************************************************************/
/*                                                                       */
/* File Name:Erc80Handler.c                                              */
/* Compiler:Metaware C                                                   */
/* Date:04/30/89                                                         */
/* Author:A. Coleman/J. Crook                                            */
/*                                                                       */
/* This program is a generic Erc 80 handler.                             */
/*                                                                       */
/* Bind this object module with FautlHander.obj and LogMessage.obj to*/
/* create ErcHandler.run.                                                */
/*                                                                       */
/*************************************************************************/

#define HEX       0x10

#define CheckErc
#define Delay
#define ErrorExit
#define FatalError
#define FProtectedMode
#define PutPointer
#define SetTrapHandler
#define WriteBsRecord
#define Syslit

#include <stdlib.h>
#include <string.h>
#include <erc.h>
#include <ctoslib.h>

typedef void (*PF) ();/* pointer to function */

extern LogMessage ();
extern void _fault_handler ();/* our assembly function */
extern bsVid[];


PF p_fault_handler = 0;/* pointer to fault handler */
Pointer BadCSIP = NULL;/* used to store address where fault was */

/*************************************************************************/
/*                                                                       */
/* set_fault_handler                                                     */
/*                                                                       */
/* This procedure is the protection fault handler.  The input            */
/* parameter is a pointer to the function that handles fault             */
/* recovery or exit.                                                     */
/*************************************************************************/
```

```
PF set_fault_handler (PF handler)
   {
   PF    old_handler;
   ErcType erc;

   /* check to see if we are operating in protected mode */
   if ((FProtectedMode ()) == FALSE ) return 0;

   /* save old fault handler so we can restore it later */
   old_handler = p_fault_handler;
   p_fault_handler = handler;

   /* set the new handler as the erc 80 handler for this program */
   CheckErc (SetTrapHandler (13, (Pointer) _fault_handler));
   return (old_handler);
   }

/*************************************************************************/
/*                                                                       */
/* handle_fault                                                          */
/*                                                                       */
/* This procedure deals with the protection fault.                       */
/*                                                                       */
/*************************************************************************/

void handle_fault()
   {
   Word cb, erc;
   char pMsgLog[] = "Erc 80 (Protection Fault) in ErcHandler.run";
   char pMsgVid[] = "Bad Pointer at CS:IP:  ";

   /* tell 'em about it */
   CheckErc (WriteBsRecord (bsVid, pMsgVid, strlen (pMsgVid), &cb));
   PutPointer (BadCSIP, HEX);

   do { /* keep trying to output the message to the log */
      erc = LogMessage (&pMsgLog, strlen(pMsgLog) );
      if (erc == 290) Delay(1); /* delay then retry */
      } while(erc == 290);

   /* now exit */
   ErrorExit(80);
   }

/*************************************************************************/
/*                                                                       */
/* string_length                                                         */
/*                                                                       */
/* This procedure returns the length of the string input parameter       */
/*                                                                       */
/*************************************************************************/

Word string_length(char *pstr)
   {
   Wordcb = 0;

   while( *pstr != 0)
      {
      pstr++;
      cb++;
      }
```

```
   return(cb);
   }

/*************************************************************************/
/*                                                                       */
/* CheckEofErc                                                           */
/*                                                                       */
/* This procedure indicates whether an EOF occurred.                     */
/*                                                                       */
/*************************************************************************/

CheckEofErc (ErcType erc)
   {
   if(erc == ercOK) return(FALSE);
   else if(erc == ercEOF) return(TRUE);
   elseFatalError(erc);
   }

/*************************************************************************/
/*                                                                       */
/*  main                                                                 */
/*                                                                       */
/*************************************************************************/

void main()
   {
   PF    OldHandler;
   char  *pstrDeath;
   Word  cb;

   pstrDeath = NULL;

   /* set address of Handle Fault as erc 80 handler */
   OldHandler = set_fault_handler (&handle_fault);

   /* try to get length from a null pointer. Causes erc 80 */
   cb = string_length (pstrDeath);

   /* if we don't use a null pointer, this happens. */
   CheckErc (WriteBsRecord (bsVid, pstrDeath, cb, &cb));

   set_fault_handler (OldHandler);
   }
```

FaultHandler.asm

This sample program is referenced by Listing 3–3.

```
; FaultHandler.asm -- generic GP fault handler
;

extrn ErrorExit:far

DGroup group Data
assume ds: DGroup

Data segment word public 'Data'
```

```
extrn p_fault_handler:WORD;tell the Assembler about p_fault_handler
extrn BadCSIP:WORD;tell it about CSIP pointer
Data ends

Stack segment stack 'stack'
     db   4    dup(?)
Stack ends

FaultHandler segment word 'code'
assume cs: FaultHandler

public _Fault_Handler
_Fault_Handler proc far

     mov  ax, dgroup
     mov  ds, ax
     mov  ax, ds:word ptr p_fault_handler
     mov  dx, ds:word ptr p_fault_handler + 2  ;load p_fault_handler
     or   ax, dx                       ;is p_fault_handler NULL?
     jz   exit                         ;if it is, exit
     mov  bx,sp                        ;else
     mov  cx,ss:[bx+2]                 ;get CS:IP of fault
     mov  BadCSIP,cx
     mov  cx,ss:[bx+4]
     mov  BadCSIP+2,cx

     jmp  dword ptr p_fault_handler  ;then jump to fault handler

exit:
     mov  ax, 80
     push ax
     call ErrorExit

_Fault_Handler endp
FaultHandler ends
end ; FaultHandler.asm
```

LogMessage.c

This sample program is referenced by Listing 3–4.

```
/*************************************************************************/
/*                                                                       */
/* File Name:LogMessage.c                                                */
/* Compiler:Metaware C                                                   */
/* Date:04/30/89                                                         */
/* Author:A. Coleman/J. Crook                                            */
/*                                                                       */
/* This module logs a text message to the system log.                    */
/*                                                                       */
/*************************************************************************/


#define WriteLog

#include <ctoslib.h>
#include <string.h>
```

```
Word LogMessage (Pointer pbMessage, Word cbMessage)
   {
   struct
     {
     Word TextCode; /* value to indicate text msg to Plog */
     Word cbText;
     Byte rgbText[120];
     } LogMsg;

   LogMsg.TextCode = 0xFFF7; /* ASCII message code */
   LogMsg.cbText = cbMessage;
   memcpy (&(LogMsg.rgbText), pbMessage, cbMessage);
   return (WriteLog (&LogMsg, cbMessage + 4));
   }
```

Chapter 4

Chapter 4 contains the following listing.

Params.c

This sample program is referenced by Listings 4–1 and 4–3.

```
/*************************************************************************/
/*                                                                       */
/* File Name:Params.c                                                    */
/* Compiler:Metaware C                                                   */
/* Date:05/05/89                                                         */
/* Author:A. Coleman/J. Crook                                            */
/*                                                                       */
/* This program accepts input parameters from the Executive when the     */
/* Run command is issued.  It then prompts the user for additional       */
/* parameters and chains to itself using the new parameters.             */
/*                                                                       */
/* Bytestreams are used rather than mixing byte streams and VAM          */
/* because it is easier keeping track of cursors with bytestreams        */
/* when using printf's (as these utilize bytestreams).                   */
/*                                                                       */
/*************************************************************************/

/* define constants */
#define ALLOC_SIZE     0x100
#define STRSIZE        20
#define MAXPARAMS      12

#define EXECFRAME      0
#define PRIORITY       0x40
#define ESCAPE         0xFF

#define CURSOR         'C'

#define FINISH         0x04
#define CANCEL         0x07
#define RETURN         0x0A
#define GO             0x1B

#define ERCNOPARAM     2440
#define ERCNOSUBPARAM  2450

#define Syslit

/* define CTOS procedures to be used */
#define AllocMemoryLL
#define Beep
#define Chain
#define CheckErc
#define CParams
#define CSubParams
#define ErrorExit
#define InitVidFrame
#define QueryVidBs
#define ReadKbd
```

```
#define ResetFrame
#define ResetMemoryLL
#define RgParam
#define RgParamInit
#define RgParamSetEltNext
#define RgParamSetListStart
#define TextEdit
#define WriteBsRecord

#include <ctoslib.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/* define the pre-opened byte streams */
externchar bsVid[];

typedef struct {
        char    iFrame;
        char    nLines;
        char    nCols;
        char    iLine;
        char    iCol;
        char    fCurVisible;
        char    fPause;
        char    bAttr;
        char    fLiteral;
        char    rgbReserved[7];
        } VidStatusType;

/* pointer and size */
typedef struct {
        char _far *pb;
        short   cb;
        } pbcbtype;

typedef struct {
        char _far *prgch;
        Word    cchMax;
        Word    cchMac;
        Word    ichCursor;
        } TdbType;

char    rgStr[STRSIZE];   /* general purpose string */

TdbType  textBlock;
TdbType  *tbPtr;
VidStatusType    strVidStatus;

char pChainFile[] = "Params.run";

char pMsgChaining[] =
        "\n\n  Chaining ...";
char pMsgIntro[] =
        "To exit this program, press FINISH at any time.";
char pMsgParam[] =
        "\nType a subparameter value, then press RETURN:  ";

/* forward procedure declarations */
int GetString (char *pStr, char *pPrompt);
```

```
/*************************************************************************/
/*                                                                       */
/* DisplayTextBuffer                                                     */
/*                                                                       */
/* Displays the text entered by the user in the TextEdit                 */
/* buffer.  The cursor must be reset after the string is written back*/
/* to the original location because the next call may overwrite this */
/* string (dependent on the editing desired).                            */
/*                                                                       */
/*************************************************************************/

void DisplayTextBuffer (Word iCol, Word iLine)
   {
   char  rgbVid [4];
   Word  cbRet;

   /* display our string */
   CheckErc (WriteBsRecord (bsVid, textBlock.prgch, textBlock.cchMax,
                            &cbRet));

   /* set up array for video escape */
   rgbVid [0] = ESCAPE;
   rgbVid [1] = CURSOR;
   rgbVid [2] = iCol;
   rgbVid [3] = iLine;

   /* reset the cursor */
   CheckErc (WriteBsRecord (bsVid, rgbVid, sizeof(rgbVid), &cbRet));
   }

/*************************************************************************/
/*                                                                       */
/* GetNewParams                                                          */
/*                                                                       */
/* This procedure prompts the user for the entry of a parameter. A GO*/
/* enables the user to enter subparams for the specified parameter.  */
/* A RETURN creates a null parameter. A CANCEL completes the VLPB and*/
/* causes a chain back to this program with the entered parameters. A*/
/* FINISH exits the program.                                             */
/*                                                                       */
/* Each string entered results in a new subparameter.                    */
/*                                                                       */
/*************************************************************************/

void GetNewParams()
   {
   char  bIn, fSkip;
   int   cbStr;
   pbcbtype pParam;
   Word  iParam = 1, jParam = 0;

   while (iParam <= MAXPARAMS)
      {
      printf ("\n\nPress GO to enter parameter %d, RETURN to skip, "
              "CANCEL to Chain, or FINISH.\n", iParam);
      fSkip = FALSE;

      /* loop until we get one of the above characters */
      while (TRUE)
         {
         CheckErc (ReadKbd (&bIn));
         if (bIn == FINISH) ErrorExit(0);
```

```
            else if (bIn == CANCEL) return;
            else if (bIn == GO) break;
            else if (bIn == RETURN)
               {
               fSkip = TRUE;
               break;
               }
            else CheckErc (Beep ());
            }

         /* we don't want to skip this parameter, collect subparameters */
         if (fSkip != TRUE)
            {

            /* initialize the subparameter list for this parameter */
            CheckErc (RgParamSetListStart (iParam));
            while (jParam <= MAXPARAMS)
               {

               /* get the next subparameter */
               cbStr = GetString(rgStr, pMsgParam);
               if (cbStr > 0)
                  {
                  pParam.pb = rgStr;
                  pParam.cb = cbStr;

                  /* store it in the VLPB */
                  CheckErc (RgParamSetEltNext (&pParam));
                  jParam++;
                  }

               /* we do not have a subparameter entry */
               else break;
               }  /* end of while jParam */
            }  /* end of if fSkip */
         iParam++;
         }  /* end of while iParam */
      }

/*********************************************************************/
/*                                                                   */
/* GetString                                                         */
/*                                                                   */
/* This procedure performs the input of the parameter string from the*/
/* user.  The cursor position must be retreived from the video status*/
/* buffer to restore it after text editing the string.               */
/*                                                                   */
/*********************************************************************/

int GetString (char *pStr, char *pPrompt)
   {
   char  i = 0;
   char  bIn;
   ErcType erc;
   Word  iCol, iLine;

   /* clear the string */
   while (i < STRSIZE) pStr[i++] = 0;

   /* set up buffer for TextEdit */
   tbPtr = &textBlock;
   textBlock.prgch= pStr;
```

```
    textBlock.cchMax= STRSIZE;
    textBlock.cchMac= 0;
    textBlock.ichCursor= 0;

    /* prompt for a parameter */
    printf (pPrompt);

    /* get the cursor postion before we enter and echo characters */
    CheckErc (QueryVidBs (bsVid, &strVidStatus));
    iCol   = strVidStatus.iCol;
    iLine  = strVidStatus.iLine;

    while(TRUE)
       {
       CheckErc (ReadKbd (&bIn));
       switch (bIn)
          {
       case FINISH:
          ErrorExit (0);
       case CANCEL:
          return (-1);
       case GO:
       case RETURN:
          return (textBlock.cchMac);
       default:
          erc = TextEdit (bIn, tbPtr);
          if (erc != 0)
             {
             CheckErc (Beep());
             break;
             }
          DisplayTextBuffer (iCol, iLine);
          break;
          } /* end of switch */
       } /* end of while */
    return(0);
    }

/*************************************************************************/
/*                                                                       */
/* ListParams                                                            */
/*                                                                       */
/* This procedure pulls out each subparam from the variable length   */
/* parameter block.  The routine stops when a No Such Parameter error*/
/* is returned.                                                          */
/*                                                                       */
/* Parameter 0's subparameters are not looked at to not get confused */
/* with the retype buffer (displaying it would not look good).       */
/*                                                                       */
/*************************************************************************/

void ListParams ()
   {
   char  i = 0, fOut = 0;
   ErcType erc;
   pbcbtype pParam;
   Word  cParam, cSubParam; /* count of params */
   Word  iParam, jParam;/* parameter indices */

   /* get input param count */
   cParam = CParams ();
```

```
    /* scan through each parameter, forget parameter 0 */
    for (iParam = 1; iParam < cParam; iParam++)
       {

       /* get the count of subparameters for this parameter */
       cSubParam = CSubParams (iParam);
       if (cSubParam > 0)
          {
          fOut = TRUE;
          printf("\n\nParameter %d has %d subparameter(s).",
                 iParam, cSubParam);

          /* if the subparam count is greater than 0, print each one */
          for (jParam = 0; jParam < cSubParam; jParam++)
             {
             erc =  RgParam (iParam, jParam, &pParam);

             /* quit if no parameter, break if no subparameter */
             if (erc == ERCNOPARAM) return;
             else if (erc == ERCNOSUBPARAM) break;
             else CheckErc (erc);

             for (i = 0; i < pParam.cb ; i++)
                rgStr[i] = *(pParam.pb + i);
             rgStr[i] = 0;
             printf(
                 "\nThe entry in parameter %d, subparameter %d is '%s'",
                 iParam, jParam, rgStr);
             }  /* end of for jParam */
          }
       }  /* end of for iParam*/
    /* we have no parameters passed to us (other than parameter 0) */
    if (fOut != TRUE)
       printf ("\n\nNo parameters passed to %s.", pChainFile);
    }

void main() {
    Pointer pVLPB;      /* used to build new VLPB */

    /* initialize the video */
    CheckErc (ResetFrame (EXECFRAME));

    printf (pMsgIntro);

    /* display all the parameters passed to me */
    ListParams ();

    /* reset long-lived memory, then allocate a new VLPB */
    CheckErc (ResetMemoryLL ());
    CheckErc (AllocMemoryLL (ALLOC_SIZE, &pVLPB));

    /* Init the parameter block */
    CheckErc (RgParamInit (pVLPB, ALLOC_SIZE, (MAXPARAMS-1)));

    /* fill it in */
    GetNewParams ();

    /* tell the user I am chaining to myself */
    printf (pMsgChaining);
    CheckErc (Chain (pChainFile, strlen(pChainFile),
                     NULL, 0, PRIORITY, 0, FALSE));
    }
```

Chapter 5

Chapter 5 contains the following listing.

SegAccess.c

This sample program is referenced by Listings 5–1 and 5–3.

```
/*************************************************************************/
/*                                                                       */
/* File Name: SegAccess.c                                                */
/* Compiler:  Metaware C                                                 */
/* Date:      05/10/89                                                   */
/* Author:    A. Coleman/J. Crook                                        */
/*                                                                       */
/* This program takes a data pointer, creates an alias pointer in the*/
/* program's Local Descriptor Table (LDT).  At the end, the data     */
/* pointer and the alias pointer are compared to verify they referenc*/
/* the same physical location.                                           */
/*                                                                       */
/* The CTOS/Open documentation for FComparePointer states that a TRUE*/
/* is returned if two pointers reference the same physical location, */
/* that TRUE is not equal to a 'C' TRUE (0x0FF vs. 1) so a check for */
/* not false is made rather than a check for a true condition.       */
/*                                                                       */
/* A union is used in this program because casting of structures to  */
/* pointers is not allowed.  With the union, we redieine a pointer as*/
/* a structure of two words so we can play with both the selector and*/
/* the offset to construct our own specific pointer.                     */
/*                                                                       */
/*************************************************************************/

#define SysLit
#define CheckErc
#define CreateAlias
#define ErrorExit
#define FComparePointer
#define SetSegmentAccess

#include <ctoslib.h>
#include <stdlib.h>
#include <stdio.h>
#include <String.h>

Pointer  pData;
Selector sgCode;
Word   offset;

union{
     POINTERWords PtrStr;
     Pointer   ptr;
     } u;

char pTestMsg [] =
     "\"This is a test\"\n";

char pCodeMsg [] =
```

```
    "The string referenced by the code pointer = ";
char pDataMsg [] =
    "The string referenced by the data pointer = ";

char pDiffMsg []  =
 "\nAlias Code Pointer and Data Pointer reference different locations";
char pSameMsg []  =
 "\nAlias Code Pointer and Data Pointer reference the same location";

void main ()
   {

   /* get the offset of our Test Message array within DS */
   pData = &pTestMsg;
   offset = offsetof (pData);
   printf (pDataMsg);
   printf (pData);

   /* now we want an alias based on the start of our DS */
   offsetof (pData) = 0;

   /* create the alias and define it as executable code */
   CheckErc (CreateAlias (pData, &sgCode));
   CheckErc (SetSegmentAccess (sgCode, 10));

   /* now construct a pointer to our code and print the string */
   u.PtrStr.ra = offset;
   u.PtrStr.sa = sgCode;
   printf (pCodeMsg);
   printf (u.ptr);

   /* finally compare the data pointer and the code pointer */
   pData = &pTestMsg;
   if (FComparePointer (pData, u.ptr, 1) != FALSE)
      printf (pSameMsg);
   else
      printf (pDiffMsg);
   ErrorExit (0);
   }
```

Chapter 6

Chapter 6 contains the following listings.

AsNeededMsg.c

This sample program is referenced by Listings 6–2 and 6–3.

```
/*************************************************************************/
/*                                                                       */
/* File Name:AsNeededMsg.c                                               */
/* Compiler:Metaware C                                                   */
/* Date:05/13/89                                                         */
/* Author:A. Coleman/J. Crook                                            */
/*                                                                       */
/* This program uses message files, however, it loads the message as     */
/* needed rather than preloading.                                        */
/*                                                                       */
/*************************************************************************/

#define CheckErc
#define ErrorExitString
#define ErrorExit
#define GetUCB
#define InitMsgFile
#define PrintMsg
#define ReadDirSector
#define WriteBsRecord
#define WriteByte

#define Syslit
#define sdType

#include <ctoslib.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#define ERCEOF              1
#define ERCINSUFFMEMORY     400
#define ERCMSGNOTFOUND      13503

#define FINISH              0x04
#define CANCEL              0x07
#define RETURN              0x0A
#define NIL                 0x0
#define MSGMAX              5
#define MAXMSGLENGTH        80
#define MAXINT              0x0FFFF

#define MSGFILECOUNT        56004

typedef struct { Word Reserverd; char sbVol[13]; char sbDir[13];
                 } ucbType;

extern char *bsVid;
```

```
char pMFileName [] ="SampleMsg.bin";
char pMsgMFileError [] ="Cannot open message file.";
char pMsgMNotFound [] ="Message not found.";

ucbType  Ucb;

/* define the message info structure and the message number array */

/* define the message and cache buffers, these must be word aligned */
Word rgMsgBuffer [512];
Word rgMsgCache [1024];

/* define the directory page buffer */
char rgbDirBuffer [512];

/* define the directory name and file name arrays */
char rgbDirectory [13];charrgbFile[80];

/*************************************************************************/
/*                                                                       */
/* CheckMsgErc                                                           */
/*                                                                       */
/* This procedure validates the error condition when using the           */
/* message routines.  A message not found condition displays an          */
/* internal message, all other non-zero errors result in a               */
/* program abort.                                                        */
/*                                                                       */
/*************************************************************************/

void CheckMsgErc (ErcType erc)
   {

   if (erc == ERCMSGNOTFOUND)
      {
      /* display the internal message not found string */
      printf ("\n");
      printf (pMsgMNotFound);
      }
   else CheckErc (erc);
   }

/*************************************************************************/
/*                                                                       */
/* GetFileCount                                                          */
/*                                                                       */
/* This procedure gets the default directory spec from the               */
/* User Control Block.  It then attempts to read the directory pages     */
/* finding how many files are present in the directory.  Each file       */
/* is listed out followed by a message indicating the total count of     */
/* files in the directory.  The message has two macros for expansion:    */
/* the name of the directory and the count of files.                     */
/*                                                                       */
/*************************************************************************/

Word GetFileCount ()
   {
   ErcType erc = ercOK;
   Pointer pb;
   Word  cb, i, iSector = 0, nFile = 0;

   /* get the ucb so we can get the default path and store it */
   CheckErc (GetUCB (&Ucb, sizeof (Ucb)));
```

```
    memset (rgbDirectory, NIL, sizeof (rgbDirectory));
    cb = Ucb.sbDir[0];
    pb = &Ucb.sbDir[1];
    memcpy (rgbDirectory, pb, cb);

    for (iSector = 0; iSector < MAXINT; iSector++)
       {

       /* read a directory sector, stop when hit EOF */
       erc = ReadDirSector (rgbDirectory, strlen(rgbDirectory), NULL, 0,
                            iSector, &rgbDirBuffer);
       if (erc == ERCEOF) break;
       else CheckErc (erc);

       /* scan sector, an entry is sbtype and file header number */
       i = 1;
       while (i < 512)
          {

          /* first check for another entry in directory page */
          if (rgbDirBuffer[i] == 0) break;

          /* get the file name */
          memset (rgbFile, NIL, sizeof (rgbFile));

          cb = rgbDirBuffer[i];
          pb = &rgbDirBuffer[i+1];
          memcpy (rgbFile, pb, cb);

          /* list the file */
          printf (rgbFile);
          printf ("\n");

          /* increment file count and adjust scan index */
          nFile = nFile + 1;

          /* add length byte and file header index to the entry index */
          i += cb + 3;
          }
       }
    return (nFile);
    }

/*************************************************************************/
/*                                                                       */
/* InitMsgEnvironment                                                    */
/*                                                                       */
/* This procedure initializes the message handling environment.  The */
/* name of the message file, the message buffer and the message cache*/
/* are all needed.  If we have an error, then we have to use one of  */
/* our internal messages.                                                */
/*                                                                       */
/*************************************************************************/

void InitMsgEnvironment ()
   {
   ErcType erc;

   /* initialize the message environment */
   erc = InitMsgFile (pMFileName, strlen(pMFileName), 0, 0,
                      &rgMsgBuffer, sizeof(rgMsgBuffer),
                      &rgMsgCache, sizeof(rgMsgCache));
```

```
   /* if we have a problem, exit with an error */
   if (erc != ercOK)
      ErrorExitString(erc, &pMsgMFileError, strlen(pMsgMFileError));
   }


void main()
   {
   sdType rgSdMsg [2]; /* sized to max parameters in msg */
   Word  nFile;

   /* initialize the message handling environment */
   InitMsgEnvironment ();

   /* get the count of files in the local directory */
   nFile = GetFileCount ();

   /* set up message parameter block */
   rgSdMsg[0].pb = (char _far *) &nFile;
   rgSdMsg[0].cb = sizeof(nFile);
   rgSdMsg[1].pb = (char _far *) &rgbDirectory;
   rgSdMsg[1].cb = strlen(rgbDirectory);

   /* print out the file count message, system expands it */
   printf ("\n");
   CheckMsgErc(PrintMsg(MSGFILECOUNT,&rgSdMsg,sizeof(rgSdMsg),&bsVid));

   ErrorExit (0);
   }
```

PreLoadMsg.c

This sample program is referenced by Listing 6–4.

```
/*************************************************************************/
/*                                                                       */
/* File Name:PreLoadMsg.c                                                */
/* Compiler:Metaware C                                                   */
/* Date:05/13/89                                                         */
/* Author:A. Coleman/J. Crook                                            */
/*                                                                       */
/* This program preloads and uses message files.                         */
/*                                                                       */
/*************************************************************************/

#define Beep
#define CheckErc
#define ErrorExitString
#define ErrorExit
#define ExpandLocalMsg
#define GetDateTime
#define GetMsgUnexpanded
#define InitMsgFile
#define ReadKbd
#define WriteBsRecord
#define WriteByte
```

```
#define Syslit
#define sdType

#include <ctoslib.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#define ERCOK               0
#define ERCINSUFFMEMORY     400
#define ERCMSGNOTFOUND      13503

#define FINISH              0x04
#define CANCEL              0x07
#define RETURN              0x0A
#define MSGMAX              5
#define MAXMSGLENGTH        80

#define MSGPRESSFINISH      56001
#define MSGDATETIME         56002

typedef struct { Pointer pb; Word cb; } pbcbType;

extern char *bsVid;

char pMFileName [] ="SampleMsg.bin";
char pMsgMFileError [] ="Cannot open message file.";
char pMsgMNotFound [] ="Message not found.";

pbcbType pbcbMacro;

/* define the message info structure and the message number array */
Word nMsg;
struct{ Word iMsg; sdType sdMsg; } rgMsgInfo [MSGMAX];
static Word rgMsgNum [] = { MSGPRESSFINISH, MSGDATETIME };

/* define the message and cache buffers, these must be word aligned */
Word rgMsgBuffer [512];
Word rgMsgCache [1024];

/* define the temporary and expansion buffers */
char rgbTempBuffer [MAXMSGLENGTH];
char rgbExpBuffer [MAXMSGLENGTH];
/*****************************************************************************/
/*                                                                           */
/* CheckMsgErc                                                               */
/*                                                                           */
/* This procedure validates error conditions when using the message          */
/* routines.  A message not found condition displays an internal             */
/* message, all other non-zero errors result in a program abort.             */
/*                                                                           */
/*****************************************************************************/

void CheckMsgErc (ErcType erc)
   {
   Word cbRet;

   if (erc == ERCMSGNOTFOUND)
      {
      /* display the internal message not found string */
      CheckErc (WriteByte (&bsVid, RETURN));
      CheckErc(WriteBsRecord(&bsVid,pMsgMNotFound,
```

```
                              strlen(pMsgMNotFound), &cbRet));
      ]
   else CheckErc (erc);
   ]

/*********************************************************************/
/*                                                                   */
/* DisplayMessage                                                    */
/*                                                                   */
/* This procedure expands the specified message and then displays it.*/
/*                                                                   */
/*********************************************************************/

void DisplayMessage (Word wMsg)
   [
   Pointer pbMsg;
   Word cbMsg, cbRet, iMsg;

   /* get the index of the proper message in the message structure */
   for (iMsg = 0; iMsg < nMsg; iMsg++)
      if (wMsg == rgMsgNum[iMsg]) break;

   /* check to see if we did not find the message */
   if (iMsg == nMsg) CheckMsgErc (ERCMSGNOTFOUND);

   /* use a temporary pointer and count to store message info */
   pbMsg = rgMsgInfo[iMsg].sdMsg.pb;
   cbMsg = rgMsgInfo[iMsg].sdMsg.cb;

   /* expand the message */
   CheckMsgErc (ExpandLocalMsg (&pbcbMacro, sizeof(pbcbMacro),
                                pbMsg, cbMsg, &rgbExpBuffer,
                                sizeof(rgbExpBuffer), &cbRet, FALSE));

   /* now display the message */
   CheckErc (WriteByte (&bsVid, RETURN));
   CheckErc (WriteBsRecord (&bsVid, &rgbExpBuffer, cbRet, &cbRet));
   CheckErc (WriteByte (&bsVid, RETURN));
   ]

/*********************************************************************/
/*                                                                   */
/* InitMsgEnvironment                                                */
/*                                                                   */
/* This procedure initializes the message handling environment.  The */
/* name of the message file, the message buffer and the message cache*/
/* are all needed.  If we have an error, then we have to use one of  */
/* our internal messages.                                            */
/*                                                                   */
/*********************************************************************/

void InitMsgEnvironment ()
   [
   ErcType    erc;
   Pointer    pbMsg;
   Word  cbMsg, iMsg;

   /* initialize the message environment */
   erc = InitMsgFile (pMFileName, strlen(pMFileName), 0, 0,
                      &rgMsgBuffer, sizeof(rgMsgBuffer),
                      &rgMsgCache, sizeof(rgMsgCache));
```

```
    /* if we have a problem, exit with an error */
    if (erc != ERCOK)
       ErrorExitString(erc, &pMsgMFileError, strlen(pMsgMFileError));

    /* get the count of messages we have */
    nMsg = sizeof (rgMsgNum) / 2;
    for (iMsg = 0; iMsg < nMsg; iMsg++)
       {

       /* retrieve a messages */
       CheckMsgErc (GetMsgUnexpanded (rgMsgNum[iMsg], &rgbTempBuffer,
                                      MAXMSGLENGTH, &cbMsg));

       /* allocate some memory to hold the unexpanded message */
       if ((pbMsg = malloc(cbMsg+1)) == NULL)
          ErrorExit (ERCINSUFFMEMORY);

       /* transfer the message into the allocated area */
       memcpy (pbMsg, rgbTempBuffer, cbMsg);

       /* now update the message info structure */
       rgMsgInfo[iMsg].iMsg = rgMsgNum[iMsg];
       rgMsgInfo[iMsg].sdMsg.pb = pbMsg;
       rgMsgInfo[iMsg].sdMsg.cb = cbMsg;
       }
    }


void main()
   {
   char  bKey;
   long  DateTime;

   InitMsgEnvironment ();

   while (TRUE)
      {

      /* first of all, set the date/time Macro expansion structure */
      pbcbMacro.pb = &DateTime;
      pbcbMacro.cb = sizeof (DateTime);

      /* get the system date and time */
      CheckErc (GetDateTime (&DateTime));

      /* display the date/time message and insert the date/time */
      DisplayMessage (MSGDATETIME);

      /* ask user to press FINISH to get out, or CANCEL to continue */
      DisplayMessage (MSGPRESSFINISH);

      while (TRUE)
         {
         CheckErc (ReadKbd (&bKey));
         if (bKey == FINISH) ErrorExit (0);
         else if (bKey == CANCEL) break;
         else CheckErc (Beep ());
         }
      }
   }
```

ServerMsgs.c

This sample program is referenced by Listing 6–4.

```
/*********************************************************************/
/*                                                                   */
/* File Name:ServerMsgs.c                                            */
/* Compiler:Metaware C                                               */
/* Date:05/13/89                                                     */
/* Author:A. Coleman/J. Crook                                        */
/*                                                                   */
/* Simple example of a program that uses server message files.       */
/*                                                                   */
/*********************************************************************/
#define  CheckErc
#define  CloseServerMsgFile
#define  ErrorExitString
#define  Exit
#define  FatalError
#define  GetDateTime
#define  GetDirStatus
#define  GetServerMsg
#define  InitMsgFile
#define  OpenServerMsgFile
#define  PrintMsg
#define  ReadKbd
#define  Syslit
#define  WriteBsRecord
#define  WriteByte
#define  sdType

#include <ctoslib.h>
#include <string.h>
#include <stdlib.h>

#define  ercOK    0
#define  Finish   0x04 /* filled box */
#define  Go       0x1B /* filled circle */
#define  NewLine  0x0A

extern   char bsVid[];
extern   FlagType fDevelopement;

char     rgbCantAccessMsgFile[] = "Cannot open message file.";
char     rgbMsgNotFound[] = "Message not found.";

InitServerMsgEnvironment(Pointer pbMsgFileName, Word cbMsgFileName)
[
ErcType erc;

erc = OpenServerMsgFile(pbMsgFileName, cbMsgFileName, 0, 0, FALSE);
if(erc != ercOK)
    ErrorExitString(erc, &rgbCantAccessMsgFile,
                    strlen(rgbCantAccessMsgFile) );
return(erc);
]

MsgErc(ErcType erc)
[
Word i;
```

```
if( erc != ercOK) {
    WriteByte(bsVid, NewLine);
    erc = WriteBsRecord(bsVid, &rgbMsgNotFound,
                        strlen(rgbMsgNotFound), &i);
    }
return(erc);
}


main()
{
sdType   sdMsgRet;
Word     iMsgNum = 56100;
Word     cbRet;
char     inkey;
char     rgMsgFileName[] = "ServerMsg.bin";

fDevelopement = TRUE;

InitServerMsgEnvironment(&rgMsgFileName, strlen(rgMsgFileName) );

MsgErc(GetServerMsg(iMsgNum, &sdMsgRet) );

WriteByte(&bsVid, NewLine);
WriteBsRecord(&bsVid, sdMsgRet.pb, sdMsgRet.cb, &cbRet);

iMsgNum = 56101;
MsgErc(GetServerMsg(iMsgNum, &sdMsgRet) );

WriteByte(&bsVid, NewLine);
WriteBsRecord(&bsVid, sdMsgRet.pb, sdMsgRet.cb, &cbRet);

CheckErc(ReadKbd(&inkey) );

return(0);
}
```

SampleMsg.txt

This file is not referenced in a listing. It is the text file that contains the messages used by the previous examples.

```
:56000: "Press %K9 to confirm %K8, or press %K4 to continue."
:56001: "Press %K8 to exit, or any other key to continue."
:56002: "The date and time are: %0D5"
:56003: "All %0N messages retrieved."
:56004: "There are %0N files in the directory named %1S."
```

Chapter 7

Chapter 7 contains the following listings.

The sample program referenced by Listing 7-1 is VamText.c. That program appears in the listings for Chapter 2, "Basic Input and Output."

The sample program referenced by Listing 7-2 is CoedProc.c. That program appears in the listings for Chapter 12, "Writing Multi-Process Programs."

Chapter 8

Chapter 8 contains the following listings.

FooServerRq.c

This sample program is referenced by Listings 8–1 through 8–7.

```
/*************************************************************************/
/*                                                                       */
/* File Name:FooServerRq.C                                               */
/* Compiler:Metaware C                                                   */
/* Date:05/11/89                                                         */
/* Author:A. Coleman/J. Crook                                            */
/*                                                                       */
/* This is a sample connection-oriented system service that returns      */
/* data in a field for a program.  The service uses Wait to receive      */
/* requests.  No queueing of requests is performed, the order is         */
/* first-in-first-out.                                                   */
/*                                                                       */
/* Note - this program must be linked with LogMessage.obj.               */
/*                                                                       */
/*************************************************************************/

#define Syslit
#define RqHeaderType
#define rqInfoType

#include "ctostypes.h"

#define AllocExch
#define CheckErc
#define ChangePriority
#define ConvertToSys
#define ChangePriority
#define Check
#define Crash
#define Delay
#define ErrorExit
#define ErrorExitString
#define ExitAndRemove
#define FatalError
#define ForwardRequest
#define GetUserNumber
#define GetPartitionHandle
#define QueryRequestInfo
#define RemovePartition
#define RequestDirect
#define Respond
#define Send
#define SetMsgRet
#define SetPartitionLock
#define SetPartitionName
#define ServeRq
#define VacatePartition
#define Wait
```

```
#include <ctoslib.h>
#include <string.h>
#include <stdlib.h>

/* opt out not needed library routines to conserve space */
#include <server.h>

#define RqDATA          0xEF00
#define RqDEINST        0xEF01
#define RqOPEN          0xEF02
#define RqCLOSE         0xEF03
#define RqSWAP          0xEF04
#define RqTERMINATE     0xEF05

#define MAXCONNECTIONS 5

#define ERCINCONSISTENCY    3
#define ERCINVALIDHANDLE    63004
#define ERCNOHANDLES        63005
#define ERCOPENCONNECTIONS  63006


typedefstruct {
     RqHeaderType RqHead;
     Word      wHandle;
     Pointer   pbDataRet;
     Word      cbDataRet;
     Pointer   psDataRet;
     Word      ssDataRet;
     } RqDataType;

typedefstruct {
     RqHeaderType RqHead;
     Word      wSysInfo;
     } SystemRqType;

typedefstruct {
     RqHeaderType RqHead;
     Word      wHandle;
     } CloseRqType;

typedefstruct {
     RqHeaderType RqHead;
     Word      *pHandleRet;
     Word      sHandleRet;
     } OpenRqType;

rqInfoType rqInfo[10];

Word srqInfo = 4;
Word rgHandles[MAXCONNECTIONS];
Word cOpenConnections = 0;
Word erc;
Word exchServe;
Word usernum;
Word phRet;
Word nRequests;

static Word rgRequestTable [] =
     { RqSWAP, RqTERMINATE, RqDATA, RqDEINST, RqCLOSE, RqOPEN };

char pServerName [] ="Foo Server";
```

```
char pMsgInstalledOk [] ="Foo Server Installed";
char pMsgInstallError [] ="Unable to install the Foo Server";
char pMsgPrevInstall [] ="Foo Server Already Installed";

char pMsgError [] ="Foo Server Deinstalled due to error";
char pMsgRet [] ="I'm the Foo Server, and I'm OK.\n";


/* External Procedures */
externLogMessage (Pointer, Word);

/* Forward Procedure Declarations */
void ProcessRequest (Pointer pRq);
void ProcessSwapRequest (SystemRqType *pRq);
void ProcessTerminationRequest (SystemRqType *pRq);
void TerminateIfErcNotOk (Word erc, RqHeaderType *pRq);

/*****************************************************************/
/*                                                               */
/* CrashIfErcNotOk                                               */
/*                                                               */
/* This procedure is used to crash the sytem if an error occurs which*/
/* we can't recover from once we are a system service.           */
/*                                                               */
/*****************************************************************/

void CrashIfErcNotOk (Word erc)
   {
   if (erc != ercOK) Crash (erc);
   }

/*****************************************************************/
/*                                                               */
/* CloseConnection                                               */
/*                                                               */
/* This procedure closes a connection with the user. An assumption is*/
/* made that all outstanding requests have already been served.  */
/*                                                               */
/*****************************************************************/

void CloseConnection (CloseRqType *pRq)
   {

   /* check to see if the user matches a value in the handle array */
   if (rgHandles[pRq->wHandle] == pRq->RqHead.userNum)
      {

      /* reset the handle and decrement the connection count */
      rgHandles[pRq->wHandle] = 0;
      cOpenConnections--;
      pRq->RqHead.ercRet = 0;
      }
   else pRq->RqHead.ercRet = ERCINVALIDHANDLE;
   }

/*****************************************************************/
/*                                                               */
/* DeInstall                                                     */
/*                                                               */
/* This procedure checks for open connections.  Pending messages on  */
/* the local service exchange are processed.  The partition is then  */
```

```
/* unlocked and the server exits.                                          */
/*                                                                         */
/*************************************************************************/

void DeInstall (pRq)
RqHeaderType *pRq;
   {
   RqHeaderType   *pWaitingRq;
   int   i;

   /* pRq is null unless a deinstall request was received */
   if (pRq != NULL)

      /* check to see if we have open connections */
      if (cOpenConnections > 0)
         {
         pRq->ercRet = ERCOPENCONNECTIONS;
         return;
         }

   /* restore the requests */
   for (i = 0; i < nRequests; i++)
      CrashIfErcNotOk (ServeRq (rgRequestTable[i], rqInfo[i].exch));

   /* loop here to clear all messages on this exchange */
   while (Check (exchServe, &pWaitingRq) == ercOK)
      ProcessRequest (pWaitingRq);

   /* all done, unlock partition, and deinstall  */
   CrashIfErcNotOk (SetPartitionLock (FALSE));
   CrashIfErcNotOk (Respond (pRq));
   ExitAndRemove ( );

   /* if we get here, there is something wrong with the OS */
   Crash (ERCINCONSISTENCY);
   }

/*************************************************************************/
/*                                                                       */
/* OpenConnection                                                        */
/*                                                                       */
/* This procedure searches for an available handle and assigns it to */
/* the user issuing the open request, it then returns the handle.    */
/*                                                                       */
/*************************************************************************/

void OpenConnection (OpenRqType *pRq)
   {
   Word wHandle;

   /* find an available handle */
   wHandle = 1;
   while ((rgHandles[wHandle] != 0) && (wHandle < MAXCONNECTIONS))
      wHandle++;

   /* check to see if all the handles are in use */
   if (wHandle == MAXCONNECTIONS)
      pRq->RqHead.ercRet = ERCNOHANDLES;
   else
      {

      /* put the handle number in the returned handle field */
```

```
      *pRq->pHandleRet = wHandle;

      /* put client's usernum in the array at position wHandle */
      rgHandles[wHandle] = pRq->RqHead.userNum;

      /* return ercOK and increment connection count */
      pRq->RqHead.ercRet = 0;
      cOpenConnections++;
      }
   }

/*************************************************************************/
/*                                                                       */
/* preTerminateIfErcNotOk                                                */
/*                                                                       */
/* This procedure is used to clean up and exit if an error               */
/* which we can recover from occurs after we are a system service.       */
/*                                                                       */
/*************************************************************************/

void preTerminateIfErcNotOk (Word erc)
   {
   int i;

   if (erc != ercOK)
      {
      LogMessage (pMsgError, strlen(pMsgError));

      /* restore the requests to their initial states */
      for (i = 0; i < nRequests; i++)
         CrashIfErcNotOk (ServeRq (rgRequestTable[i], rqInfo[i].exch));

      /* unlock my partition and get out */
      CrashIfErcNotOk (SetPartitionLock (FALSE));
      ExitAndRemove ();

      /* OS problem if we ever get here */
      Crash (ERCINCONSISTENCY);
      }
   }

/*************************************************************************/
/*                                                                       */
/* ProcessDataRequest                                                    */
/*                                                                       */
/* This procedure processes all the data requests which come in.         */
/*                                                                       */
/*************************************************************************/

void ProcessDataRequest (RqDataType *pRq)
   {
   Word i = 0;

   /* check for valid handle */
   if (pRq->RqHead.userNum != rgHandles[(pRq->wHandle)])
      {
      pRq->RqHead.ercRet = ERCINVALIDHANDLE;
      return;
      }

   /* calculate the maximum string size */
   if (pRq->cbDataRet < strlen (pMsgRet))
```

```
      i = pRq->cbDataRet;
   else
      i = strlen (pMsgRet);

   /* put the message and the size in the user's buffer */
   memcpy (pRq->pbDataRet, &pMsgRet, i);
   *((Word *)(pRq->psDataRet)) = i;

   /* return an error OK condition */
   pRq->RqHead.ercRet = 0;
   }

/*********************************************************************/
/*                                                                   */
/* ProcessRequest                                                    */
/*                                                                   */
/* This procedure preprocesses all the requests which come in.       */
/*                                                                   */
/*********************************************************************/

void ProcessRequest (Pointer pRq)
   {

   /* identify request, then take appropriate action */
   switch( ((RqHeaderType *) pRq)->rqCode)
      {
   case RqDEINST:
      DeInstall(pRq);
      break;
   case RqDATA:
      ProcessDataRequest (pRq);
      break;
   case RqOPEN:
      OpenConnection (pRq);
      break;
   case RqSWAP:
      ProcessSwapRequest(pRq);
      break;
      case RqTERMINATE:
      ProcessTerminationRequest(pRq);
      break;
   case RqCLOSE:
      CloseConnection(pRq);
      break;
   default:
      TerminateIfErcNotOk(31, NULL); /* "No such request code" */
      }
   }

/*********************************************************************/
/*                                                                   */
/* ProcessSwapRequest                                                */
/*                                                                   */
/* This procedure tells the OS that it is OK to swap the user out.   */
/* It assumes that all outstanding requests have already been served.*/
/*                                                                   */
/*********************************************************************/

void ProcessSwapRequest (SystemRqType *pRq)
   {

   /* just return an error OK condition */
```

```
   pRq->RqHead.ercRet = 0;
   }

/*************************************************************************/
/*                                                                       */
/* ProcessTermination Request                                            */
/*                                                                       */
/* This procedure tells the OS that it is OK to terminate the user.      */
/* It assumes that all outstanding requests have already been served.    */
/*                                                                       */
/*************************************************************************/

void ProcessTerminationRequest (SystemRqType *pRq)
   {
   Word i;

   /* remove all the users open connections */
   for (i = 0; i < MAXCONNECTIONS; i++)
      if (rgHandles[i] == pRq->RqHead.userNum)
         {
         rgHandles[i] = 0;
         cOpenConnections--;
         }

   /* return an error OK condition */
   pRq->RqHead.ercRet = 0;
   }

/*************************************************************************/
/*                                                                       */
/* TerminateIfErcNotOk                                                   */
/*                                                                       */
/* This procedure is used to clean up the environment when an error      */
/* occurs which does not prevent us from deinstalling ourselves.         */
/*                                                                       */
/*************************************************************************/

void TerminateIfErcNotOk (Word erc, RqHeaderType *pRq)
   {

   if (erc != ercOK)
      {
      LogMessage(pMsgError, strlen(pMsgError));

      /* check to see if we do not have a null pointer */
      if (pRq != NULL) pRq->ercRet = 33; /* service not available */

      /* now attempt to deinstall */
      DeInstall (pRq);
      }
   }

/*************************************************************************/
/*                                                                       */
/* Main                                                                  */
/*                                                                       */
/* Set up the partition as a server.  First, though, we have to check    */
/* to see if another copy of the server is running.  If one is, we       */
/* must terminate; otherwise proceed with the installation.              */
/*                                                                       */
/*************************************************************************/
```

```
void main ()
   {
   int   i;
   RqHeaderType *pRq;

   /* first get my user number, I need this later */
   CheckErc (GetUserNumber (&usernum));

   /* allocate the exchange I will be using */
   CheckErc(AllocExch(&exchServe));

   /* check to see if another copy of me is running */
   erc = GetPartitionHandle (pServerName, strlen(pServerName), &phRet);
   if (erc == ercOK)
      ErrorExitString(0, pMsgPrevInstall, strlen(pMsgPrevInstall));

   /* query the requests, if any are undefined, then get out */
   nRequests = sizeof(rgRequestTable) / 2;
   for (i = 0; i < nRequests; i++)
      CheckErc(QueryRequestInfo(rgRequestTable[i],&rqInfo[i],srqInfo));

   /* inform the user I've installed */
   CheckErc (SetMsgRet (pMsgInstalledOk, strlen(pMsgInstalledOk)));
   erc = ConvertToSys ();
   if (erc != 0)
      ErrorExitString(erc, pMsgInstallError, strlen(pMsgInstallError));
   else ErrorExit (0);


   /* Servers must run at priority less than 40 to prevent deadlock */
   preTerminateIfErcNotOk (ChangePriority (12));


   /* Set the partition name so deinstallation can occur */
   preTerminateIfErcNotOk (SetPartitionName (0, pServerName,
                                             strlen(pServerName)));

   /* Now serve the requests */
   for (i = 0; i < nRequests; i++)
      preTerminateIfErcNotOk (ServeRq (rgRequestTable[i], exchServe));

   /* now loop waiting for a request, serve it when I get one */
   while (TRUE)
      {
      /* wait for an incoming request */
      CrashIfErcNotOk (Wait (exchServe, &pRq));
      /* process the request */
      ProcessRequest(pRq);
      /* now respond to the user */
      CrashIfErcNotOk (Respond (pRq));
      }
   }
```

FooServerRequest.txt

This sample program is referenced by Listing 8–8.

```
:WsAbortRq: 0EF05h
:TerminationRq:0EF05h
:SwappingRq: 0EF04h
:ChgUserNumRq: 0EF06h

:RequestCode: 0EF00h
:RequestName: GetFooText
:Version:  2
: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: NoRouting

:RequestCode: 0EF01h
:RequestName: DeinstallFooServer
:Version:  1
:LclSvcCode: 0000h
:ServiceExch: exchInstalledMastr
:sCntInfo:  0
:nReqPbCb:  0
:nRespPbcb:  0
:Params:  none
:NetRouting: noRouting
:SrpRouting: NoRouting

:RequestCode: 0EF02h
:RequestName: OpenFooServer
:Version:  1
:LclSvcCode: 0000h
:ServiceExch: exchInstalledMastr
:sCntInfo:  0
:nReqPbCb:  0
:nRespPbcb:  1
:Params:  p(12), c(2,16)
:NetRouting: OpenFh,CloseAtTermination
:SrpRouting: NoRouting

:RequestCode: 0EF03h
:RequestName: CloseFooServer
:Version:  1
:LclSvcCode: 0000h
:ServiceExch: exchInstalledMastr
:sCntInfo:  2
:nReqPbCb:  0
:nRespPbcb:  0
:Params:  w(12)
:NetRouting: CloseFh,rFh
:SrpRouting: NoRouting

:RequestCode: 0EF04h
:RequestName: SwapClientFooServer
:Version:  1
```

```
:LclSvcCode: 0000h
:ServiceExch: exchInstalledMastr
:sCntInfo:  2
:nReqPbCb:  0
:nRespPbcb:  0
:Params:  none
:NetRouting: noRouting
:SrpRouting: NoRouting

:RequestCode: 0EF05h
:RequestName: TerminateClientFooServer
:Version:  1
:LclSvcCode: 0000h
:ServiceExch: exchInstalledMastr
:sCntInfo:  2
:nReqPbCb:  0
:nRespPbcb:  0
:Params:  none
:NetRouting: noRouting
:SrpRouting: NoRouting

:RequestCode: 0EF06h
:RequestName: ChangeUserClientFooServer
:Version:  1
:LclSvcCode: 0000h
:ServiceExch: exchInstalledMastr
:sCntInfo:  2
:nReqPbCb:  0
:nRespPbcb:  0
:Params:  none
:NetRouting: noRouting
:SrpRouting: NoRouting
```

FooServerRqLabel.asm

This sample program is referenced by Listing 8–9.

```
; request labels for the requests in FooServerRq.c
; FooServerRqLabel.asm

$INCLUDE([Sys]<Sys>RqLabl.mdf)

%RqName( 0EF00h, "GetFooText")
%RqName( 0EF01h, "DeinstallFooServer")
%RqName( 0EF02h, "OpenFooServer")
%RqName( 0EF03h, "CloseFooServer")
```

QueryFooServerRq.c

This sample program is not referenced. It sends a request to the Foo service and displays the response.

```
/*****************************************************************/
/*                                                               */
```

```
/* File Name: QueryFooServerRq.c                                        */
/* Compiler:  Metaware C                                                */
/* Date:      05/11/89                                                  */
/* Author:    A. Coleman/J. Crook                                       */
/*                                                                      */
/* This is a sample connection-oriented program which opens a           */
/* connection with a system service.  It then loops receiving           */
/* messages from the service.  The program closes the connection        */
/* when any keyboard action is taken.                                   */
/* Link with FooServerRqLabel.obj.                                      */
/*                                                                      */
/************************************************************************/

#define CheckErc
#define Delay
#define ErrorExit
#define OutputToVid0
#define ReadKbdDirect

#include <ctoslib.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 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 the server.\n\n";

void main ()
   {

   /* open my connection with the foo server */
   CheckErc( OpenFooServer (&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);
}
```

Chapter 9

Chapter 9 contains the following listings.

FooServerSc.c

This sample program is referenced by Listings 9–1, 9–2, and 9–3.

```
/*************************************************************************/
/*                                                                       */
/* File Name:FooServerSc.C                                               */
/* Compiler:Metaware C                                                   */
/* Date:05/11/89                                                         */
/* Author:A. Coleman/J. Crook                                            */
/*                                                                       */
/* This is a sample connectionless system common service that returns*/
/* data in a field for a program.                                        */
/*                                                                       */
/* Note - this program must be compiled LARGE so that DS is pushed   */
/*        on entry to GetFooText.                                        */
/*                                                                       */
/* Note - the object module must be linked with LogMessage.obj.      */
/*                                                                       */
/*************************************************************************/

#define Syslit
#define RqHeaderType
#define rqInfoType

#include <ctostypes.h>

#define AllocExch
#define CheckErc
#define ChangePriority
#define ConvertToSys
#define ChangePriority
#define Check
#define Crash
#define Delay
#define ErrorExit
#define ErrorExitString
#define Exit
#define ExitAndRemove
#define FatalError
#define ForwardRequest
#define GetUserNumber
#define GetPartitionHandle
#define QueryRequestInfo
#define RemovePartition
#define RequestDirect
#define Respond
#define Send
#define SetMsgRet
#define SetPartitionLock
#define SetPartitionName
#define ServeRq
#define SystemCommonQuery
```

```
#define SystemCommonInstall
#define VacatePartition
#define Wait

#include <ctoslib.h>
#include <string.h>
#include <stdlib.h>

/* opt out not needed library routines to conserve space */
#include <server.h>

#define ERCINCONSISTENCY    3
#define RqDEINST            0x0EF01
#define ScID                26005

/* external procedures */
extern LogMessage(char *, int);

/* forward procedures */
void   DeInstall();

struct rgSysComInfo {
         Byte fDefined;
         Pointer pProc;
         Word orgbParamDef;
         Word srgbParamDef;
         Byte rgbParamDef[20];
         } rgSysComInfo;

rqInfoType rqInfo;

Word sSysComInfo = 29;
Word erc;
Word exchServe;
Word usernum;
Word phRet;

char pServerName[] ="Foo SysCom";

char pMsgInstalledOk [] ="Foo Server Installed";
char pMsgInstallError [] ="Unable to install the Foo Server";
char pMsgPrevInstall [] ="Foo Server Already Installed";
char pMsgScInUse [] ="System Common Number already in Use";

char pMsgError [] ="Foo Server Deinstalled due to error";
char pMsgRet [] ="I'm the Foo Server, and I'm OK.\n";

char pParams[] = "pcp";


/*********************************************************************/
/*                                                                   */
/* CrashIfErcNotOk                                                   */
/*                                                                   */
/* This procedure is used to crash the sytem if an error occurs which*/
/* we can't recover from once we are a system service.               */
/*                                                                   */
/*********************************************************************/

void CrashIfErcNotOk (Word erc)
   {
   if (erc != ercOK) Crash (erc);
```

```
   }

/*************************************************************************/
/*                                                                       */
/* DeInstall                                                             */
/*                                                                       */
/* This procedure restores the requests and system common procedures */
/* to their initial values.  The partition is then unlocked and the  */
/* server exits.                                                         */
/*                                                                       */
/*************************************************************************/

void DeInstall (pRq)
RqHeaderType *pRq;
   [

   /* restore the requests */
   CrashIfErcNotOk (ServeRq (RqDEINST, rqInfo.exch));

   /* restore the system commons */
   CrashIfErcNotOk (SystemCommonInstall (ScID, rgSysComInfo.pProc,
                                         rgSysComInfo.rgbParamDef,
                                         rgSysComInfo.srgbParamDef, 1,
                                         NULL, 0) );

   /* All done, unlock partition, and deinstall  */
   CrashIfErcNotOk (SetPartitionLock (FALSE));
   CrashIfErcNotOk (Respond (pRq));
   ExitAndRemove ();

   /* if we get here, something is wrong with the OS */
   Crash(ERCINCONSISTENCY);
   ]

/*************************************************************************/
/*                                                                       */
/* GetFooText                                                            */
/*                                                                       */
/* This procedure processes calls to the system common procedure from*/
/* client processes.                                                     */
/*                                                                       */
/*************************************************************************/

/* Don't use the C calling convention, use CTOS style */
pragma Calling_convention(CTOS_CALLING_CONVENTIONS);

ErcType GetFooText(Pointer pbDataRet,Word cbDataRet, Pointer psDataRet)
   [
   Word i;

   /* get the maximum size of the message */
   if (cbDataRet < strlen (pMsgRet))
      i = cbDataRet;
   else
      i = strlen (pMsgRet);

   /* put the message in the user's buffer */
   memcpy (pbDataRet, pMsgRet, i);
   *((Word *) psDataRet) = i;

   /* return ercOK */
   return(0);
```

```
   }

/* restore C calling convention */
pragma Calling_convention();

/*************************************************************************/
/*                                                                       */
/* preTerminateIfErcNotOk                                                */
/*                                                                       */
/* This procedure is used to clean up and exit if an error               */
/* which we can recover from occurs after we are a system service.       */
/*                                                                       */
/*************************************************************************/

void preTerminateIfErcNotOk (Word erc)
   {
   if (erc != ercOK)
      {
      LogMessage (pMsgError, strlen(pMsgError));

      /* restore the requests */
      CrashIfErcNotOk (ServeRq (RqDEINST, rqInfo.exch));

      /* restore the system commons */
      CrashIfErcNotOk (SystemCommonInstall (ScID, rgSysComInfo.pProc,
                                            rgSysComInfo.rgbParamDef,
                                            rgSysComInfo.srgbParamDef, 1,
                                            NULL, 0) );

      /* unlock my partition and get out */
      CrashIfErcNotOk (SetPartitionLock (FALSE));
      ExitAndRemove ();

      /* OS problem if we ever get here */
      Crash (ERCINCONSISTENCY);
      }
   }

/*************************************************************************/
/*                                                                       */
/* TerminateIfErcNotOk                                                   */
/*                                                                       */
/* This procedure is used to clean up the environment when an error      */
/* occurs which does not prevent us from deinstalling ourselves.         */
/*                                                                       */
/*************************************************************************/

void TerminateIfErcNotOk(erc, pRq)
Word     erc;
RqHeaderType *pRq;
   {
   if (erc != ercOK)
      {
      LogMessage (pMsgError, strlen(pMsgError));

      /* check to see if we do not have a null pointer */
      if (pRq != NULL) pRq->ercRet = 33; /* service not available */

      /* now attempt to deinstall */
      DeInstall(pRq);
      }
   }
```

```
/*********************************************************************/
/*                                                                   */
/* Main                                                              */
/*                                                                   */
/* Set up the partition as a server.  First, though, we have to check*/
/* to see if another copy of the server is running.  If one is, we   */
/* must terminate; otherwise proceed with the installation.          */
/*                                                                   */
/*********************************************************************/

void main ()
   {
   RqHeaderType *pRq;

   /* first get my user number, I need this later */
   CheckErc (GetUserNumber (&usernum));

   /* allocate the exchange I will be using */
   CheckErc(AllocExch(&exchServe));

   /* check to see if another copy of me is running */
   erc = GetPartitionHandle (pServerName, strlen(pServerName), &phRet);
   if (erc == ercOK)
      ErrorExitString(0, pMsgPrevInstall, strlen(pMsgPrevInstall));

   /* query the system common procedure. If it's in use, error exit. */
   CheckErc (SystemCommonQuery (ScID, &rgSysComInfo, sSysComInfo));
   if (rgSysComInfo.fDefined != FALSE)
      ErrorExitString(0, pMsgScInUse, strlen(pMsgScInUse));

   /* query the request.  If it is in use, error exit. */
   CheckErc (QueryRequestInfo(RqDEINST, &rqInfo, sizeof(rqInfo)));

   /* inform the user I've installed */
   CheckErc (SetMsgRet (pMsgInstalledOk, strlen(pMsgInstalledOk)));
   erc = ConvertToSys ();
   if (erc != 0)
      ErrorExitString (erc,pMsgInstallError, strlen(pMsgInstallError));
   else ErrorExit (0);

   /* Servers must run at a priority below 40 to prevent a deadlock */
   preTerminateIfErcNotOk (ChangePriority (12));

   /* Set the partition name so deinstallation can occur */
   preTerminateIfErcNotOk (SetPartitionName (0, pServerName,
                                             strlen(pServerName)));

   /* Now serve the requests */
   preTerminateIfErcNotOk (ServeRq (RqDEINST, exchServe));

   /* Now server the system common */
   preTerminateIfErcNotOk(SystemCommonInstall(ScID,
                                 (Pointer) GetFooText, pParams,
                                 strlen(pParams), 0, NULL, 0));

   /* now loop waiting for a request, serve it when I get one */
   while (TRUE)
      {
      Wait (exchServe, &pRq);

      /* the only one I service is a deinstallation request */
```

```
        DeInstall (pRq);
        }
    }
```

FooSysCommonLabel.asm

This sample program is referenced by Listings 4–1 and 4–3.

```
; FooSysCommonLabel.asm  Sys common proc label for FooServerSc.c
STACK SEGMENT STACK 'STACK'     ; default stack so programs won't crash!
    DB 512 DUP (?)
STACK ENDS

$include([Sys]<Sys>ossub.mdf)

%OsSubLab(26005, "GetFooText")
```

QueryFooServerSc.c

This sample program is not referenced. It calls the Foo system common service.

```
/*************************************************************************/
/*                                                                       */
/* File Name: QueryFooServerSc.c                                         */
/* Compiler:  Metaware C                                                 */
/* Date:      05/11/89                                                   */
/* Author:    A. Coleman/J. Crook                                        */
/*                                                                       */
/* This is a sample program which issues a system common call to         */
/* receive a message to print. The program terminates when any           */
/* keyboard action is taken.                                             */
/* Link with FooSysCommonLabel.obj.                                      */
/*                                                                       */
/*************************************************************************/

#define CheckErc
#define Delay
#define ErrorExit
#define OutputToVid0
#define ReadKbdDirect

#include <ctoslib.h>
#include <string.h>

pragma Calling_convention(CTOS_CALLING_CONVENTIONS);

extern ErcType GetFooText (Pointer pbDataRet, Word cbDataRet,
                           Pointer psDataRet);

pragma Calling_convention();

#define CHECKONLY      1
#define DELAYRATE      10
```

```
#define ERCNOCHARAVAIL 602

Word cbRet;

char bKey;
char rgbServerData[80];

char pMsgIntro [] =
        "Press any key to discontinue use of the Foo Server.\n\n";

void main ()
   {

   /* display the intro message */
   OutputToVid0 (pMsgIntro, strlen(pMsgIntro));

   while (ReadKbdDirect (CHECKONLY, &bKey) == ERCNOCHARAVAIL)
      {

      /* get a message from the server */
      CheckErc (GetFooText (&rgbServerData, sizeof (rgbServerData),
                            &cbRet));

      /* display the message */
      OutputToVid0 (&rgbServerData, cbRet);

      /* now wait awhile before continuing */
      CheckErc (Delay (DELAYRATE));
      }
   ErrorExit (0);
   }
```

Chapter 10

Chapter 10 contains no listings.

Chapter 11

Chapter 11 contains the following listings.

Mediator.asm

This sample program is referenced by Listing 11-1.

```
; Filename:  Mediator.asm.   Implements a mediator for WriteBsRecord

extrn WriteBsRecord:far

Data segment word public 'Data'
Data ends

Stack segment stack 'stack'
     db   24   dup(?)
Stack ends

DGroup group Data
assume ds: DGroup
assume ss: DGroup

MedWriteBsRecord segment word 'code'
assume cs: MedWriteBsRecord

; on entry stack has:           Needs to be:
;    bp+18     sn(pcbRet)         sn(pBSWA)
;    bp+16     o(pcbRet)          o(pBSWA)
;    bp+14     cb                 sn(pb)
;    bp+12     sn(pb)             o(pb)
;    bp+10     o(pb)              cb
;    bp+8      sn(pBSWA)          sn(pcbRet)
;    bp+6      o(pBSWA)           o(pcbRet)

; We need to reverse the positions of the parameters

public MedWriteBsRecord
public _MedWriteBsRecord
_MedWriteBsRecord proc far
     push bp
     mov  bp,sp

     push [bp+8]   ; push pBSWA, seg and offset
     push [bp+6]
```

```
    push [bp+12]  ; push pb, seg and offset
    push [bp+10]

    push [bp+14]  ; push cb

    push [bp+18]  ; push pcbRet, seg and offset
    push [bp+16]

    call WriteBsRecord ; make the call

    mov  sp,bp
    pop  bp
    ret           ;don't pop original stack.  Caller does it.
_MedWriteBsRecord endp
MedWriteBsRecord ends
end; Mediator.asm
```

TestMediator.c

This sample program is not referenced in the text. It exercises Mediator.asm.

```
/*************************************************************************/
/*                                                                       */
/* File Name: TestMediator.c                                             */
/* Compiler:  Metaware C                                                 */
/* Date:      09/6/89                                                    */
/* Author:    A. Coleman                                                 */
/*                                                                       */
/* This file tests a mediator for the C language calling                 */
/* conventions.  The CTOS procedure it mediates is WriteBsRecord.        */
/*                                                                       */
/* The mediator is in the file Mediator.asm.  It should be Linked        */
/* with this file's object module.                                       */
/*                                                                       */
/*************************************************************************/

#include <stdlib.h>
#include <string.h>

#define Syslit
#include <ctostypes.h>

/* preopened vid bytestream */
extern char *bsVid[];

/* our mediated procedure.  Note that we use C calling convention */
extern ErcType MedWriteBsRecord(Pointer pBSWA, Pointer pb, Word cb,
                                Pointer pcbRet);

char *rgbMessage = "\nThis is a test message for MedWriteBsRecord.";
Word cbMessage;
Word cbCharsRet;
ErcType erc;
```

```
main() [
cbMessage = strlen(rgbMessage);

while(TRUE)
    erc = MedWriteBsRecord(bsVid, rgbMessage, cbMessage, &cbCharsRet);

return(0);
]
```

Chapter 12

Chapter 12 contains the following listings. The three listings together form a single program, which demonstrates multiple-processes and disposable initialization code. The three modules should be linked in the following order:

main.obj multiproc.obj coedproc.obj

Main.obj must appear first so that the Linker can arrange the code, coed, and DGROUP segments of the program in the correct order.

Multiproc.c

This sample program is referenced by Listings 12–1, and 12–2.

```
/*************************************************************************/
/*                                                                       */
/* File Name:  MultiProc.c                                               */
/* Compiler:   Metaware C                                                */
/* Date:       05/16/89                                                  */
/* Author:     J. Crook                                                  */
/*                                                                       */
/* This program is an interesting little example showing the multi-      */
/* processing capabilities of CTOS.  The main process initiates three    */
/* asynchronous processes and then waits for a FINISH key depression     */
/* from the user (basically going to sleep).                             */
/*                                                                       */
/* The first child process is a clock process.  This process does        */
/* nothing more than update a time display about once a second.          */
/*                                                                       */
/* The second child process puts dots on the screen at one of two        */
/* pre-determined rates.  The third childprocess erases dots from the    */
/* screen also at two pre-determined rates.                              */
/*                                                                       */
/* Note - the stacks are in my DGroup segment rather than allocating     */
/* them from the memory pool.  This is because I want my child           */
/* processes to have addressability of data elements in DGroup.  I       */
/* allocate them using the ExpandAreaSL call by expanding my DGroup.     */
/* The run file must be bound with DS Allocation specified.              */
/*                                                                       */
/* Note - bind this with Main.obj.  This module declares the proper      */
/* ordering of the segments so that the COED segment is deallocated      */
/* correctly.  Additionally, there is no Main 'C' routine as the 'C'     */
/* run time preamble does an expansion of DGroup before the program      */
/* code is executed.  Therefore, a stub which simply jumps to            */
/* the real main routine is coded in assembly language.  Main.obj        */
/* must be the first object module in the link list.                     */
/*                                                                       */
/*************************************************************************/

#define Syslit

#define AllocExch
```

```
#define AllocMemorySL
#define Beep
#define CheckErc
#define CreateProcess
#define DeallocMemorySL
#define Delay
#define ErrorExit
#define ExpandAreaSL
#define GetDateTime
#define NlsStdFormatDateTime
#define PutFrameChars
#define QueryFrameBounds
#define ReadKbd

#include <ctoslib.h>
#include <string.h>
#include <stdio.h>

#define BLANK    0x20
#define FINISH   0x04
#define GO       0x1b
#define MAXCH    80
#define STACKSZ  1024

typedef void (*PF) ();  /* pointer to function */

typedef struct
   { PF        pEntry;
     Selector  saData;
     Selector  saExtra;
     Selector  saStack;
     Offset    oStackInit;
     Byte      priority;
     Flag      fSys;
     Word      defaultExch;
     Flag      fSuspend;
   } ProcDescType;

/* external declarations */
extern Initialize ();

char rgchTime [32];
char rgchPath [MAXCH];

long sysTime;

Word cchTime, nCols = 0, nLines = 0, cChars = 0;

char *rgchFrame;
char *rgchSpaces;

Word *rgProlStack;
Word *rgPro2Stack;
Word *rgTimeStack;

ProcDescType ProlDesc;
ProcDescType Pro2Desc;
ProcDescType TimeDesc;

char rgMsgTitle [] =
       "Multi-Tasking Example (Press FINISH to exit)";
```

```
/*************************************************************************/
/*                                                                       */
/* AdjustScreenParams                                                    */
/*                                                                       */
/*This routine updates the column and line parameters for the display*/
/* and erasure of the dots from the video. Also, the increment          */
/* flags may be modified to vary the rate of display.                    */
/*                                                                       */
/*************************************************************************/

void AdjustScreenParams (Word *piC, Word *piL, Flag *pfC, Flag *pfL)
   {

   /* update column */
   if (*pfC == TRUE) *piC = (*piC + 1);
   else *piC = (*piC + nCols - 1);
   *piC %= nCols;

   if (*piC == 0) *pfC = TRUE;
   if (*piC == nCols - 1) *pfC = FALSE;

   /* update line */
   if (*pfL == TRUE) *piL = (*piL + 1);
   else *piL = (*piL + nLines - 1);
   *piL %= nLines;

   if (*piL == 0) *pfL = TRUE;
   if (*piL == nLines - 1) *pfL = FALSE;
   }

/*************************************************************************/
/*                                                                       */
/* AllocateArrays                                                        */
/*                                                                       */
/* This routine allocates a few of the arrays I need.  A blank array */
/* is used to clear the status line when updating the clock.  A frame*/
/*cell array is used to hold count information to prohibit holes from*/
/* appearing when dot erasure is being performed.                        */
/*                                                                       */
/*************************************************************************/

void AllocateArrays ()
   {
   Offset   raData;
   Pointer  pData;
   Selector snData;
   Word  cb, i;

   /* get the limits on frame 0, needed for memory allocation */
   CheckErc (QueryFrameBounds (0, &nCols, &nLines));

   /* we must get the selector DGROUP */
   pData = &raData;
   snData = selectorof (pData);

   /* call ExpandAreaSL to get the segment limit offset */
   CheckErc (ExpandAreaSL (0, snData, &raData));

   /* now expand the data segment to its maximum size */
   cb = (nCols * nLines) + (STACKSZ * 6) + MAXCH;
   CheckErc (ExpandAreaSL (cb, snData, &raData));
```

```
   /* set the offset for the three stacks */
   for (i = 0; i < 3; i++)
      [
      switch (i)
         [
      case 0:
         selectorof (rgTimeStack) = snData;
         offsetof   (rgTimeStack) = raData;
         break;
      case 1:
         selectorof (rgProlStack) = snData;
         offsetof   (rgProlStack) = raData;
         break;
      case 2:
         selectorof (rgPro2Stack) = snData;
         offsetof   (rgPro2Stack) = raData;
         break;
         ]
      raData += (STACKSZ * 2);
      ]

   /* set the offset for the array for video frame 0 */
   selectorof (rgchFrame) = snData;
   offsetof   (rgchFrame) = raData;
   cb = nCols * nLines;
   memset (rgchFrame, 0, cb);

   /* set the offset for the spaces array */
   raData += nCols * nLines;
   selectorof (rgchSpaces) = snData;
   offsetof   (rgchSpaces) = raData;
   memset (rgchSpaces, BLANK, nCols);
   ]

/*************************************************************************/
/*                                                                       */
/* Procl                                                                 */
/*                                                                       */
/* This routine is the process for displaying dots to the video.         */
/*                                                                       */
/*************************************************************************/

void Procl ()
   {
   char ch = GO;
   Flag fIncCol = TRUE, fIncLine = TRUE;
   Word iCol = 0, iLine = 0, iCell;

   /* build the snake */
   while (TRUE)
      {
      /* put out a single dot */
      CheckErc (PutFrameChars (0, iCol, iLine, &ch, 1));

      /* increment the frame cell counter, character counter */
      iCell = iLine * nCols + iCol;
      rgchFrame [iCell]++;
      cChars ++;

      /* get ready to go to the next cell */
      AdjustScreenParams (&iCol, &iLine, &fIncCol, &fIncLine);
```

```
      /* additional delay when we are moving up to the left */
      CheckErc (Delay (1));
      if(((fIncCol != TRUE) && (fIncLine != TRUE)) || (cChars > MAXCH))
         CheckErc (Delay (1));

      }
   }

/*************************************************************************/
/*                                                                       */
/* Proc2                                                                 */
/*                                                                       */
/* This routine is the process for erasing dots to the video.            */
/*                                                                       */
/*************************************************************************/

void Proc2 ()
   {
   char ch = BLANK;
   Flag fIncCol = TRUE, fIncLine = TRUE;
   Word iCol = 0, iLine = 0, iCell;

   /* shrink the snake */
   while (TRUE)
      {
      /* we can erase a cell with a cell count = 1, next decrement */
      iCell = iLine * nCols + iCol;
      if (rgchFrame [iCell] == 1)
         CheckErc (PutFrameChars (0, iCol, iLine, &ch, 1));
      rgchFrame [iCell]--;
      cChars--;

      /* get ready to go to the next cell */
      AdjustScreenParams (&iCol, &iLine, &fIncCol, &fIncLine);

      /* additional delay when we are not moving up to the left */
      CheckErc (Delay (1));
      if(((fIncCol == TRUE) || (fIncLine == TRUE)) && (cChars < MAXCH))
         CheckErc (Delay (1));
      }
   }

/*************************************************************************/
/*                                                                       */
/* StartProcess                                                          */
/*                                                                       */
/* This routine implements a general routine for creating processes.     */
/*                                                                       */
/*************************************************************************/

void StartProcess (PF pProc, Pointer pStack, Word cbStack,
                   ProcDescType *ProcDesc)
   {
   Offset   raStack;
   Selector saStack;
   Word     exch;

   /* allocate the default response exchange for the clock process */
   CheckErc (AllocExch (&exch));

   /* get the selector and offset of the stack */
   saStack = selectorof (pStack);
```

```
    raStack = offsetof   (pStack);

    /* set up the process descriptor */
    ProcDesc->pEntry      = pProc;
    ProcDesc->saData      = saStack;
    ProcDesc->saExtra     = 0;
    ProcDesc->saStack     = saStack;
    ProcDesc->oStackInit  = raStack + cbStack;
    ProcDesc->priority    = 0x7F;
    ProcDesc->fSys        = 0;
    ProcDesc->defaultExch = exch;
    ProcDesc->fSuspend    = 0x00;

    /* create the process */
    CheckErc (CreateProcess (ProcDesc));
    }

/*************************************************************************/
/*                                                                       */
/* TimeProc                                                              */
/*                                                                       */
/* This routine is the process for updating the time display             */
/*                                                                       */
/*************************************************************************/

void TimeProc ()
    {
    Word cSpaces, iCol;

    while (TRUE)
        {

        /* get the system date and time, expand and then display it */
        CheckErc (GetDateTime (&sysTime));
        CheckErc (NlsStdFormatDateTime (NULL, 0xFFFF, sysTime, &rgchTime,
                                        sizeof (rgchTime), &cchTime));
        CheckErc(PutFrameChars(1, nCols-cchTime, 1, &rgchTime, cchTime));

        /* now fill in the blank space from the path to the time */
        iCol = strlen (rgchPath);
        cSpaces = nCols - cchTime - iCol;
        CheckErc (PutFrameChars (1, iCol, 1, rgchSpaces, cSpaces));

        /* now wait awhile */
        CheckErc (Delay (10)); /* set to 10, for 1 second  */
        }
    }

void StartIt ()
    {
    char  ch;
    Pointer pInit;

    /* initialize everything */
    Initialize ();

    /* now dispose of the COED segment */
    pInit = (Pointer *) Initialize;
    offsetof (pInit) = 0;
    CheckErc (DeallocMemorySL (pInit, 0xFFFF));

    /* now allocate all my arrays */
```

```
    AllocateArrays ();

    /* start the clock */
    StartProcess (&TimeProc, rgTimeStack, (STACKSZ*2), &TimeDesc);

    /* start my display processes */
    StartProcess (&Proc1, rgProlStack, (STACKSZ*2), &ProlDesc);
    StartProcess (&Proc2, rgPro2Stack, (STACKSZ*2), &Pro2Desc);

    /* wait for a FINISH key, beep if any other key pressed */
    while (TRUE)
       [
       CheckErc (ReadKbd (&ch));
       if (ch == FINISH) ErrorExit (0);
       CheckErc (Beep ());
       ]

    ErrorExit (0);
    ]
```

Coedproc.c

This sample program is not referenced in the text. It performs initialization for Multiproc.c, and is then deallocated. For Multiproc.c to run properly, this file must be made into a COED segment using the ObjMunge (or similar) utility. Chapter 10, "Stack Format and Calling Conventions" gives more information about disposable initialization code.

```
/*************************************************************************/
/*                                                                       */
/* File Name:  CoedProc.c                                                */
/* Compiler:   Metaware C                                                */
/* Date:       05/16/89                                                  */
/* Author:     J. Crook                                                  */
/*                                                                       */
/* This program is the initialization code for MultiProc.run.  The       */
/* code has been put in a separate module so it can be deallocated to*/
/* shrink the code space after the program is initialized.               */
/*                                                                       */
/*************************************************************************/

/* force the code to go in the coed segment */
pragma Cgroup ("CoedGroup");
pragma Code ("Coed");

#define Syslit
#define CheckErc
#define CreateExecScreen
#define GetpStructure
#define PutFrameAttrs
#define PutFrameChars
#define QueryVidHdw
#define ResetVideo

#include <ctoslib.h>
#include <string.h>
```

```
#include <stdio.h>

#define BLANK          0x20
#define BOLD           0x10
#define MAXCH          80
#define TYPEPARTITION  36
#define SYSCONFIG      0x2C8
#define VMPARTITION    3

pragma Calling_convention(CTOS_CALLING_CONVENTIONS);
extern ErcType BuildFileSpec(Word UserNum, Pointer pbOutputSpec,
                  Pointer pcbOutputSpec, Pointer pcbOutputSpecNoPw,
                  Word cbOutputSpecMax, Byte fDefaultPath, Pointer
                  pbNode, Word cbNode, Pointer pbVol, Word cbVol,
                  Pointer pbDir, Word cbDir, Pointer pbFile, Word
                  cbFile,  Pointer pbPassword, Word cbPassword, Byte
                  fDefaultPrefix, Byte fCanonical, Byte bMode);
pragma Calling_convention();

extern char rgchPath [];
extern Word nCols, nLines;
extern char rgMsgTitle [];

/*********************************************************************/
/*                                                                   */
/* Initialize                                                        */
/*                                                                   */
/* This routine is used to initialize the entire video environment. I*/
/* am using the CreateExecScreen routine to make the display like the*/
/* Executive's screen. I am displaying the current path and a message*/
/* informing the user how to exit.                                   */
/*                                                                   */
/*********************************************************************/

void Initialize ()
   {
   Byte  rgbVidHdw[3];
   char  *pSCB;
   Word  sMap, cbNoPw, cb, ich;

   /* find out the screen size */
   CheckErc (QueryVidHdw (&rgbVidHdw, sizeof(rgbVidHdw)));
   nLines = rgbVidHdw[1];
   nCols  = rgbVidHdw[2];

   /* reset the screen, set up screen similar to the Exec */
   CheckErc (ResetVideo (nCols, nLines, TRUE, BLANK, &sMap));
   CheckErc (CreateExecScreen (sMap, nCols, nLines));

   /* display the title */
   cb = strlen (rgMsgTitle);
   CheckErc (PutFrameChars (1, 0, 0, rgMsgTitle, cb));

   /* now hi-lite the string, only if we are on CTOS/VM */
   CheckErc (GetpStructure (SYSCONFIG, 0, &pSCB));
   offsetof (pSCB) += TYPEPARTITION;
   if (*pSCB == VMPARTITION)
      CheckErc (PutFrameAttrs (1, 0, 0, BOLD, cb));

   /* start the string to display the path */
   strcpy (&rgchPath, "Path: ");
   ich = 6;
```

```
    /* build the path spec with the node, volume, and directory;   */
    /* we want the default path, so 0's are in all the path fields */
    CheckErc (BuildFileSpec (0, &rgchPath[ich], &cb, &cbNoPw,
                             MAXCH-ich, ONES, NULL, 0, NULL, 0,
                             NULL, 0, NULL, 0, NULL, 0, 0, 0, 0));
    ich += cbNoPw;

    /* null terminate the string and display it */
    rgchPath[ich] = 0;
    CheckErc (PutFrameChars (1, 0, 1, rgchPath, strlen (rgchPath)));
    }

/* braked the coed segment directive */
pragma Code ();
```

Main.asm

This sample program is not referenced in the text. It arranges the program's segments in the correct order, and bypasses the MetaWare C initialization code. This bypass is needed only for disposable initialization code.

```
;*****************************************************************************;
;*                                                                           *;
;* File Name: Main.asm                                                       *;
;* Compiler:  Assembler                                                      *;
;* Date:      05/25/89                                                       *;
;* Author:    J. Crook                                                       *;
;*                                                                           *;
;* This program is the main stub for MultiProc.run.  It predeclares          *;
;* the order of the segments so the COED segment may be deallocated          *;
;* after initialization.  Also, a stub entry routine is coded to             *;
;* prevent the 'C' run time preamble from expanding DGroup, so COED          *;
;* deallocation can be done correctly.                                       *;
;*                                                                           *;
;*****************************************************************************;

; declare my external 'Main' routine
EXTRN    StartIt:FAR

; now declare the segments and their order
Coed SEGMENT PARA PUBLIC 'Coed'
Coed ENDS
Const SEGMENT WORD PUBLIC'CONST'
Const ENDS
Data SEGMENT WORD PUBLIC'DATA'
Data ENDS
Memory SEGMENT WORD PUBLIC'MEMORY'
Memory ENDS

; declare my stack segment
Stack SEGMENT WORD PUBLIC'STACK'
    DW   200H DUP (?)
wStack EQU THIS WORD
Stack ENDS
```

```
; group all my DGroup stuff
DGroup GROUP Const, Data, Memory, Stack
; declare the code segment
Main SEGMENT WORD PUBLIC'CODE'
Main ENDS
; declare an extra segment needed for the linker ('C' run time)
_MWAHSEGMENT SEGMENT PARA PUBLIC 'AHSHIFT'
_MWAHSEGMENT ENDS

; now code the entry point
Main SEGMENT
ASSUME CS:Main
Begin:
     MOV AX, DGroup
     MOV SS, AX                        ; load DGroup into SS
ASSUME SS:DGroup
     MOV SP, OFFSET DGroup:wStack      ; initialize stack pointer
     MOV DS, AX                        ; load DGroup into DS
ASSUME DS:DGroup
     CALL StartIt                      ; go to my 'C' code
Main ENDS
END Begin
```

Chapter 13

Chapter 13 contains the following listings.

QueryFooServerAsync.c

This sample program is referenced by Listings 13–1, 13–2, and 13–3.

```
/*************************************************************************/
/*                                                                       */
/* File Name: QueryFooServerRq.Async.c                                   */
/* Compiler:  Metaware C                                                 */
/* Date:      05/11/89                                                   */
/* Author:    A. Coleman                                                 */
/*                                                                       */
/* This is a sample connection-oriented program which opens a            */
/* connection with a system service.  It then loops receiving            */
/* messages from the service.  The program closes the connection         */
/* when any keyboard action is taken.                                    */
/* This program uses the asynchronous Check primitive, allowing          */
/* it to perform other work while it waits for the response from         */
/* the service                                                           */
/*                                                                       */
/*************************************************************************/

#define AllocExch
#define Check
#define CheckErc
#define Delay
#define ErrorExit
#define GetUserNumber
#define OutputToVid0
#define PutByte
#define PutChar
#define ReadKbdDirect
#define Request

#define kernelErc
#define Syslit
#define RqHeaderType

#include <stdlib.h>
#include <stdarg.h>
#include <ctoslib.h>
#include <string.h>
#include <erc.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 pHandleRet);

pragma Calling_convention ();

ErcType BuildRequestBlock(RqHeaderType *pRq, Byte sCntInfo, Byte
```

```
            nReqPbCb, Byte nRespPbCb, Word exchResp, Word RqCode,
            char *pParamString, ...) ;

#define  CHECKONLY         1
#define  DELAYRATE         10
#define  ERCNOCHARAVAIL    602
#define  RQGETFOOTEXT      0xEF00
#define  MAXRQSIZE         64   /* 64 bytes is maximum rq block size */

PointerpRqBlockContents;
Pointer pMyRq;

Word hConnect;
Word cbRet;

char bKey;
char rgbServerData[80];

char pMsgIntro [] =
        "Press any key to close the connection with the server.\n\n";

void main ()
   {
   Word  exchMine;

   if((pMyRq = malloc(MAXRQSIZE)) == NULL)
         ErrorExit(400);

   CheckErc(AllocExch(&exchMine));

   /* open my connection with the foo server */
   CheckErc( OpenFooServer (&hConnect));

   /* display the intro message */
   OutputToVid0 (pMsgIntro, strlen(pMsgIntro));

    /* build the request block */
    CheckErc(BuildRequestBlock(pMyRq, 2, 0, 2, exchMine, RQGETFOOTEXT,
                           "%W %P %W %P %W", hConnect, &rgbServerData,
                           sizeof(rgbServerData), &cbRet, 2) );

  /* loop checking for a keyboard character, any one */
   while (ReadKbdDirect (CHECKONLY, &bKey) == ERCNOCHARAVAIL)
      {
      /* request a message from the server */
      CheckErc (Request(pMyRq) );

      /* do some other stuff while the rq is outstanding */
      while(Check(exchMine, &pMyRq) == ercNoMessage) {
           PutChar(0x2E); /* 0x2E == . */
            }

      /* display the message */
      OutputToVid0 (&rgbServerData, cbRet);

      /* now wait awhile before continuing */
      CheckErc (Delay (DELAYRATE));
      }

   /* close my connection */
   CheckErc (CloseFooServer (hConnect));
   ErrorExit (0);
```

```
    }

/*********************************************************************/
/* this procedure builds a request block.  It accepts a variable-    */
/*  length argument list, and builds the request block appropriately.*/
/*  The request-block parameter string types are:                     */
/*        %Ba Byte type                                               */
/*        %Da DoubleWord type                                         */
/*        %Oan Offset type (same as Word)                             */
/*        %Pa Pointer type                                            */
/*        %Sa Selector type                                           */
/*        %Wa Word type                                               */
/*********************************************************************/
ErcType BuildRequestBlock(RqHeaderType *pRq, Byte sCntInfo,
             Byte nReqPbCb, Byte nRespPbCb, Word exchResp,
             Word rqCode, char *pParamString, ...)
{
     va_list pArgs;/* points to each arg in turn */
     Word userNum;
     ErcType erc;

     /* get user number */
     CheckErc(GetUserNumber(&userNum));

     /* fill in request block header */
     pRq->sCntInfo = sCntInfo;
     pRq->RtCode = 0;
     pRq->nReqPbCb = nReqPbCb;
     pRq->nRespPbCb = nRespPbCb;
     pRq->userNum = userNum;
     pRq->exchResp = exchResp;
     pRq->ercRet = 0;
     pRq->rqCode = rqCode;

     /* fill in variable part */

     /* set up argument list */
     va_start(pArgs, pParamString);

     /* set pointer to first byte of request block contents */
     pRqBlockContents = pRq;
     offsetof(pRqBlockContents) += 12;

     while(*pParamString) {
          if(*pParamString != '%')
               pParamString++;
          else switch(*(++pParamString)) {
          case 'b':
          case 'B':
               *(Byte *) pRqBlockContents = va_arg(pArgs, Byte);
               offsetof(pRqBlockContents) += sizeof(Byte);
               break;
          case 'd':
          case 'D':
               *(DWord *) pRqBlockContents = va_arg(pArgs, DWord);
               offsetof(pRqBlockContents) += sizeof(DWord);
               break;
          case 'p':
          case 'P':
               *(Pointer *) pRqBlockContents = va_arg(pArgs, Pointer);
               offsetof(pRqBlockContents) += sizeof(Pointer);
```

```
                break;
            case 's':
            case 'S':
                *(Selector *) pRqBlockContents = va_arg(pArgs, Selector);
                offsetof(pRqBlockContents) += sizeof(Selector);
                break;
            case 'o':
            case 'O':
            case 'w':
            case 'W':
                *(Word *) pRqBlockContents = va_arg(pArgs, Word);
                offsetof(pRqBlockContents) += sizeof(Word);
                break;
            default:
                break;
            }
        }  /* end of while */
        return(0);
}
```

Chapter 14

Chapter 14 contains the following listings.

The listings in this section form one overlayed program. They should appear on the Linker's object module field as follows:

vcsm.obj overlay0.obj/o overlay1.obj/o overlay2.obj/o

VCSM.c

This sample program is referenced by Listing 14–1.

```
/*************************************************************************/
/*                                                                       */
/* File Name: VCSM.c                                                     */
/* Compiler:  Metaware C                                                 */
/* Date:      08/7/89                                                    */
/* Author:    S. Emmons/A. Coleman/J. Crook                              */
/*                                                                       */
/* This program shows an example of how to use Virtual Code Segment      */
/* Management calls, including:                                          */
/*                                                                       */
/*                InitOverlays                                           */
/*                EnableSwapperOptions                                   */
/*                GetCParasOvlyZone                                      */
/*                GetOvlyStats                                           */
/*                MakePermanent                                          */
/*                ReleasePermanence                                      */
/*                                                                       */
/*************************************************************************/
#include <stdio.h>

#define Syslit
#include <CTOSTypes.h>

#define AllocMemorySL
#define CheckErc
#define InitLargeOverlays
#define InitOverlays
#define EnableSwapperOptions
#define GetCParasOvlyZone
#define GetOvlyStats
#include <CTOSLib.h>

extern GetSystemInfo();
extern DoSwapperStuff();

typedef struct {
     Word cbOvMax;
     Word cbOv2ndMax;
     Word cbParMax;
     }    OverlayStatsType;

void main () {
```

```
Word                cbOvlyZone;
Word                cSwapParas;
Pointer             pSwapBuffer;
OverlayStatsType    OverlayStats;

/*
    Determine the size of our overlays
*/
CheckErc (GetOvlyStats (&OverlayStats, 6));

printf("Overlay stats:  Largest = %d, 2nd Largest = %d, Total = %d.\n",
       OverlayStats.cbOvMax, OverlayStats.cbOv2ndMax,
       OverlayStats.cbParMax * 16);

    /* convert bytes to paragraph count */
cSwapParas = (OverlayStats.cbOvMax + OverlayStats.cbOv2ndMax) >> 4;
    /* round it up to a 512-byte disk sector boundary */
cSwapParas = ((cSwapParas + 0x001F) & 0xFFE0);
    /* add an extra sector */
cSwapParas += 0x0020;

/*
    Allocate memory for the Swap Buffer;  Swap buffer needs to be
    large enough to accomodate both the largest and second largest
    overlays.
*/
CheckErc (AllocMemorySL( (cSwapParas * 16), &pSwapBuffer));

/*
    Initialize the Virtual Code Management facility
*/
CheckErc (InitLargeOverlays (pSwapBuffer, cSwapParas));

/*
    Get the size of the overlay zone (in paragraphs) and display the
    number of bytes; NOTE that 1 paragraph = 16 bytes.
*/
cbOvlyZone = (GetCParasOvlyZone () * 16);
printf ("\nThe overlay zone size is %d bytes.\n", cbOvlyZone);

/*
    Call a procedure in the first overlay
*/
printf ("\nCalling procedure GetSystemInfo in Overlay0 ... ");
GetSystemInfo();
printf("\nOne overlay in memory.");
/*
    Call a procedure in the second overlay
*/
printf ("\nCalling procedure DoSwapperStuff in Overlay1 ... ");
DoSwapperStuff ();
}
```

Overlay0.c

This sample program is not referenced in the text. It is called during execution of VCSM.c.

```
/*************************************************************************/
/*                                                                       */
/* File Name: Overlay0.c                                                 */
/* Compiler:  Metaware C                                                 */
/* Date:      08/7/89                                                    */
/* Author:    S. Emmons/A. Coleman                                       */
/*                                                                       */
/* Called by vcsm.c                                                      */
/*                                                                       */
/*************************************************************************/
#include <stdio.h>
#include <ctype.h>

#define ParDescType
#define Syslit
#define Syscom
#define SysConfigType
#include <CTOSTypes.h>

#define CheckErc
#define GetPartitionStatus
#define GetpStructure
#include <CTOSLib.h>

void GetSystemInfo () {
SysConfigType*pSysCnfgBlk;

    CheckErc (GetpStructure (ATpConfiguration, 0, &pSysCnfgBlk));
    printf ("\nSystem information follows:\n");
    printf("\nSystemBuildType: 0%0XH\t\t",
           pSysCnfgBlk->SystemBuildType);
    printf ("Operating system type: 0%0XH", pSysCnfgBlk->OsType);
    printf ("\nSaMinLL: 0%0XH\t\t\t", pSysCnfgBlk->SaMinLL);
    printf ("SaCurrLL: 0%0XH", pSysCnfgBlk->SaCurrLL);
    printf ("\nSaCurrSL: 0%0XH\t\t", pSysCnfgBlk->SaCurrSL);
    printf ("SaMaxSL: 0%0XH", pSysCnfgBlk->SaMaxSL);
    printf ("\nSaMemMax: 0%0XH\t\t", pSysCnfgBlk->SaMemMax);
    printf ("cPcb: 0%0XH", pSysCnfgBlk->cPcb);
    printf ("\ncExch: 0%0XH\t\t\t", pSysCnfgBlk->cExch);
    printf ("cLinkBlk: 0%0XH", pSysCnfgBlk->cLinkBlk);
    printf ("\ncLinkBlkRes: 0%0XH\t\t", pSysCnfgBlk->cLinkBlkRes);
    printf ("cTrb: 0%0XH", pSysCnfgBlk->cTrb);
    printf ("\ncIob: 0%0XH\t\t\t", pSysCnfgBlk->cIob);
    printf ("cFcb: 0%0XH", pSysCnfgBlk->cFcb);
    printf ("\ncVhb: 0%0XH\t\t\t", pSysCnfgBlk->cVhb);
    printf ("cUcb: 0%0XH", pSysCnfgBlk->cUcb);
    printf ("\ncUfb: 0%0XH\t\t\t", pSysCnfgBlk->cUfb);
    printf ("HardwareType: 0%0XH", pSysCnfgBlk->HardwareType);
    printf ("\nClusterConfiguration: 0%0XH\t",
```

```
            pSysCnfgBlk->ClusterConfiguration);
    printf ("fNoFileSystem: 0%0XH", pSysCnfgBlk->fNoFileSystem);
    printf ("\nfCommIOP: 0%0XH\t\t\t", pSysCnfgBlk->fCommIOP);
    printf ("fMultipartition: 0%0XH", pSysCnfgBlk->fMultipartition);
    printf ("\nnParDesc: 0%0XH\t\t\t", pSysCnfgBlk->nParDesc);
    printf ("oRgUcb: 0%0XH", pSysCnfgBlk->oRgUcb);
    printf ("\noRgOExUcb: 0%0XH\t\t", pSysCnfgBlk->oRgOExUcb);
    printf ("userNumVid: 0%0XH\n", pSysCnfgBlk->userNumVid);

}   /* end of GetSystemInfo () */

void TryToSwapIn () {
ParDescTypePartDesc;

CheckErc (GetPartitionStatus (0, 0, &PartDesc, 33));
printf("\nGot Partition Status in TryToSwapIn()");

}   /* end of TryToSwapIn () */
```

Overlay1.c

This sample program is not referenced in the text. It is called during execution of VCSM.c.

```
/************************************************************************/
/*                                                                      */
/* File Name: Overlayl.c                                                */
/* Compiler:  Metaware C                                                */
/* Date:      08/7/89                                                   */
/* Author:    S. Emmons/A. Coleman                                      */
/*                                                                      */
/* Called by vcsm.c                                                     */
/*                                                                      */
/************************************************************************/
#include <stdio.h>
#include <ctype.h>

#define Syslit
#include <CTOSTypes.h>

#define CheckErc
#define MakePermanent
#define MakePermanentP
#define MapIovlyCS
#define ReleasePermanence
#define UpdateOverlayLRU
#include <CTOSLib.h>

extern DoOtherSwapperStuff ();
extern TryToSwapIn ();

void DoSwapperStuff () {
Selector Cs;
Pointer  pProcInfo;

printf("\n2nd Overlay in memory.");
```

```
/* Make this overlay permanent. */
CheckErc (MakePermanent ());

/* Get the CS of the third overlay; Remember, zero-based, so the first
     parameter, 2, means the third overlay */
CheckErc (MapIovlyCS (2, &Cs));
selectorof (pProcInfo) = Cs;

/*   Make call to procedure in the third overlay. */
printf ("\nCalling procedure DoOtherSwapperStuff in Overlay2 ... ");
DoOtherSwapperStuff ();

/*   Make third overlay permanent. */
CheckErc (MakePermanentP (pProcInfo));

CheckErc (ReleasePermanence ());

UpdateOverlayLRU (pProcInfo);

/* Try to swap Overlay0 back into memory;  Should succeed since we did
     the ReleasePermanence call. */
printf ("\nCalling procedure TryToSwapIn in Overlay0 ... ");

TryToSwapIn ();

}    /* end of DoSwapperStuff () */
```

Overlay2.c

This sample program is not referenced in the text. It is called during execution of VCSM.c.

```
/*************************************************************************/
/*                                                                       */
/* File Name: overlay2.c                                                 */
/* Compiler:  Metaware C                                                 */
/* Date:      08/7/89                                                    */
/* Author:    S. Emmons/A. Coleman                                       */
/*                                                                       */
/* Called by vcsm.c                                                      */
/*                                                                       */
/*************************************************************************/
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define Syslit
#define UCBType
#include <CTOSTypes.h>

#define CheckErc
#define GetUCB
#include <CTOSLib.h>

void DoOtherSwapperStuff () {
UCBType  UCB;
Byte     rgbTmp[40];
```

```
printf("\nIn DoOtherSwaperStuff() - third overlay.");

CheckErc (GetUCB (&UCB, 95));

memcpy (rgbTmp, UCB.rgbVol, UCB.cbVol);
rgbTmp[UCB.cbVol] = '\0';
printf ("\n\nDefault volume: %s", rgbTmp);

memcpy (rgbTmp, UCB.rgbDir, UCB.cbDir);
rgbTmp[UCB.cbDir] = '\0';
printf ("\nDefault directory: %s", rgbTmp);

memcpy (rgbTmp, UCB.rgbPswd, UCB.cbPswd);
rgbTmp[UCB.cbPswd] = '\0';
printf ("\nDefault password: %s", rgbTmp);

memcpy (rgbTmp, UCB.rgbPrefix, UCB.cbPrefix);
rgbTmp[UCB.cbPrefix] = '\0';
printf ("\nDefault prefix: %s", rgbTmp);

memcpy (rgbTmp, UCB.rgbNode, UCB.cbNode);
rgbTmp[UCB.cbNode] = '\0';
printf ("\nDefault node: %s", rgbTmp);

}    /* end of DoOtherSwapperStuff () */
```

Chapter 15

Chapter 15 contains the following listings.

The sample program referenced by Listing 15-1 is Multiproc.c. That program appears in the listings for Chapter 12, "Writing Multi-Process Programs."

RtcTimer.c

This sample program is referenced by Listings 15–2 and 15–3.

```
/*************************************************************************/
/*                                                                       */
/* File Name: RTCTimer.c                                                 */
/* Compiler:  Metaware C                                                 */
/* Date:      06/28/89                                                   */
/* Author:    A. Coleman                                                 */
/*                                                                       */
/* This program illustrates the use of the CTOS Real-Time Clock.         */
/* It contains a procedure to initialize and start the clock.            */
/* It's main loop illustrates a good way to build timing-dependent       */
/* loops.  It ensures that all ticks are processed once and only         */
/* once.  It also minimizes the size of the timing window in which       */
/* the RTC may reset cEvents when the application is not ready for       */
/* it. Even if that happens, the algorithm recovers.                     */
/*                                                                       */
/*************************************************************************/

#define Syslit
#define ProcDescType

#define AllocExch
#define CheckErc
#define CloseRTClock
#define ErrorExit
#define GetDateTime
#define NlsFormatDateTime
#define OpenRTClock
#define Wait
#define WriteBsRecord
#define WriteByte

#define LINEFEED       0x0A
#define DateTimeLength 80
#define kernelErc
#include <erc.h>

#include <stdio.h>
#include <time.h>
#include <ctoslib.h>
#include <stdlib.h>

typedef struct TRBType {
    Word          counter, counterReload;
```

```
    short int     cEvents;
    Word          exchResp;
    ErcType       ercRet;
    Word          rqCode;
    } TRBType;

/* define the pre-opened byte streams */
externchar    bsVid[];

/* define external ssl2.0 procedure */
extern ErcType ShortDelay(Word n);

Word fPopTick;
char rgbTemplate[] = "!Www! !Nnn! !*d!, !yyyy! !*h!:!0m!:!0s! !2*A!";
Word sTemplate = 45;
char rgbTimeMsg[] = "The system time is:  ";
Word sTimeMsg = 22;

/* Build a timer block, then start the timer. */
Word InitTimer(TRBType *pTimerblock, Word cPeriods, Word rqCode)
{
    Word exchTimer;

    /* build the TRB */
    pTimerblock->cEvents = 0;
    pTimerblock->ercRet = 0;
    pTimerblock->counter = cPeriods;
    pTimerblock->counterReload = cPeriods;
    pTimerblock->rqCode = rqCode;

    /* allocate an exchange, and put it in the TRB */
    CheckErc(AllocExch(&exchTimer));
    pTimerblock->exchResp = exchTimer;

    /* open the timer. It starts ticking if counter field != 0 */
    CheckErc(OpenRTClock(pTimerblock));

    /* return the exchange for future communication with the timer */
    return(exchTimer);
}

/* Get date and time, then print it.  If we're not behind, delay a */
/* while */
void DoMyWork()
{
DateTimeType  lDateTime = 0;
char          szDateTime[DateTimeLength];
Word          cLengthRet = 0;
Word          cBytesRet = 0;
int           i;

CheckErc(GetDateTime(&lDateTime));
CheckErc(NlsFormatDateTime(NULL, lDateTime, &rgbTemplate, sTemplate,
                           &szDateTime, DateTimeLength, &cLengthRet));
if(cLengthRet < 26)
    printf("Wierd length returned by NlsFormatDateTime\n");
WriteBsRecord(bsVid, rgbTimeMsg, sTimeMsg, &cBytesRet);
WriteBsRecord(bsVid, szDateTime, cLengthRet, &cBytesRet);
WriteByte(bsVid, LINEFEED);

if(fPopTick==FALSE) {
    i = (rand() % 100);
```

```
    ShortDelay(i);
    ]
}


main()
{
Word     iTimeCount, iMaxIterations, exchMyTimer, wMyRqCode;
Word     iIterations = 0;
TRBType Timerblock;
TRBType *pTRBRet;

fPopTick = FALSE;

wMyRqCode = 0x5457;
printf("Enter time interval for RealTime Clock (1 = 100 msec):  ");
scanf("%d", &iTimeCount);
printf("\nEnter number of times RTC should fire:  ");
scanf("%d", &iMaxIterations);


    /* start the real-time clock */
exchMyTimer = InitTimer(&Timerblock, 0 /* iTimeCount */, wMyRqCode);
if(exchMyTimer == 0)
    ErrorExit(10);

    /* 7/6/89 Start Timer */
    /* Load reload field first, in case interval is short */
Timerblock.counterReload = iTimeCount;
Timerblock.counter = iTimeCount;

    /* loop forever */
while(TRUE) {
        /* wait for a message from the RTC */
    CheckErc(Wait(exchMyTimer, &pTRBRet));
    if(pTRBRet->rqCode != wMyRqCode)
        printf("Received wrong timer block from RTC.\n");
        /* do the work, then make sure we haven't missed a tick */
        /* if we have, loop until cEvents is 0. */
    do {
        DoMyWork();
        if(++iIterations >= iMaxIterations) {
            CheckErc(CloseRTClock(pTRBRet));
            return(0);
            }
        /* note:  If processor is slow, loop might not keep up with
        timer.  Would therefore drop a tick occasionally. */
        if(pTRBRet->cEvents > 1) {
            printf("Missed a tick\n");
            fPopTick = TRUE;
            }
        else
            fPopTick = FALSE;
        /* if we're up to date (cEvents == 0), go back to the Wait */
        } while(--pTRBRet->cEvents > 0);
    }
}
```

RtcTimer.Single.c

This sample program is referenced by Listing 15-4.

```
/*********************************************************************/
/*                                                                   */
/* File Name: RTCTimer.Single.c                                      */
/* Compiler:  Metaware C                                             */
/* Date:      06/28/89                                               */
/* Author:    A. Coleman                                             */
/*                                                                   */
/* This program illustrates the use of the CTOS Real-Time Clock      */
/* to time a single, rather than a repeating, interval.              */
/* It contains a procedure to initialize and start the clock.        */
/* It's main loop illustrates a good way to build timing-dependent   */
/* loops.  It ensures that all ticks are processed once and only     */
/* once.  It also minimizes the size of the timing window in which   */
/* the RTC may reset cEvents when the application is not ready for   */
/* it. Even if that happens, the algorithm recovers.                 */
/*                                                                   */
/*********************************************************************/

#define Syslit
#define ProcDescType

#define AllocExch
#define Check
#define CheckErc
#define CloseRTClock
#define DeallocExch
#define ErrorExit
#define GetDateTime
#define NlsFormatDateTime
#define OpenRTClock
#define Wait
#define WriteBsRecord
#define WriteByte

#define LINEFEED       0x0A
#define DateTimeLength 80
#define ercNoMessage   14
#define kernelErc
#include <erc.h>

#include <stdio.h>
#include <time.h>
#include <ctoslib.h>
#include <stdlib.h>

typedef struct TRBType {
    Word          counter, counterReload;
    short         int  cEvents;
    Word          exchResp;
    ErcTypeerc    Ret;
    Word          rqCode;
    } TRBType;


/* Build a timer block, then start the timer. */
Word InitTimer(TRBType *pTimerblock, Word cPeriods, Word rqCode)
```

```
[
     Word exchTimer;

     /* build the TRB */
     pTimerblock->cEvents = 0;
     pTimerblock->ercRet = 0;
     pTimerblock->counter = cPeriods;
     pTimerblock->counterReload = cPeriods;
     pTimerblock->rqCode = rqCode;

     /* allocate an exchange, and put it in the TRB */
     CheckErc(AllocExch(&exchTimer));
     pTimerblock->exchResp = exchTimer;

     CheckErc(OpenRTClock(pTimerblock));
     /* return the exchange for future communication with the timer */
     return(exchTimer);
]

/* Print a value in a static variable.  Increments with each call. */
void DoMyWork()
[
     static long i=0;
     i++;
     printf("\n i=%lu",i);
]


main()
[
Word     iTimeCount, exchMyTimer, wMyRqCode;
ErcTypeerc=0;
TRBType Timerblock;
TRBType *pTRBRet;

wMyRqCode = 0x5457;

     /* get the interval */
printf("Enter time interval for RealTime Clock (1 = 100 msec):  ");
scanf("%d", &iTimeCount);

     /* set up the real-time clock, but don't start it */
exchMyTimer = InitTimer(&Timerblock, 0, wMyRqCode);
if(exchMyTimer == 0)
     ErrorExit(10);

     /* start the clock */
Timerblock.counter = iTimeCount;

     /* loop until timer expires */
while(TRUE) [
          /* wait for a message from the RTC */
     erc = Check(exchMyTimer, &pTRBRet);
     if(erc == ercNoMessage)
          DoMyWork();
     else if(erc == 0)
          break;
     else
          CheckErc(erc);
     ]
CheckErc(CloseRTClock(pTRBRet));
CheckErc(DeallocExch(exchMyTimer));
]
```

PitTimeout.c

This sample program is referenced by Listings 15-5, 15-6, 15-7 and 15-8.

```
/*********************************************************************/
/*                                                                   */
/* File Name: PITTimeOut.c                                           */
/* Compiler:  Metaware C                                             */
/* Date:      08/5/89                                                */
/* Author:    A. Coleman                                             */
/*                                                                   */
/* This program illustrates use of the CTOS Programmable Interval    */
/* Timer.  It contains a procedure to initialize and start the timer.*/
/* It's main loop illustrates a good way to build timing-dependent   */
/* loops.  It performs work until the work is completed or until a   */
/* timeout occurs.                                                   */
/*                                                                   */
/*********************************************************************/

#define Syslit
#define ProcDescType

#define AllocExch
#define Check
#define CheckErc
#define Crash
#define ErrorExit
#define FatalError
#define FComparePointer
#define GetDateTime
#define NlsFormatDateTime
#define PSend
#define PutChar
#define PutWord
#define ResetTimerInt
#define SetTimerInt
#define Wait
#define WriteBsRecord
#define WriteByte

#define LINEFEED       0x0A
#define DateTimeLength 80
#define NUMCLIENTS     5
#define kernelErc
#include <erc.h>

#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <ctoslib.h>

typedef void _far (*pFunc)();

typedefstruct TPIBType {
    unsigned      long link1, link2;
    pFunc         pIntHandler;
    unsigned      saData;
    unsigned      cIntervals;
    Pointer       pRqBlkRet;
    unsigned      footprint;
```

```
    unsigned      delta;
    unsigned long  res1, res2;
} TPIBType;

typedef TPIBType *TPIBPtrType;

TPIBPtrType   pTimerTPIB;

/* define the pre-opened byte streams */
externchar    bsVid[];

/* public exchange for use by timer int */
Word exchPublic;
/* count of timeouts */
Word cTimeOuts;
/* number of times timer should fire */
Word      iIterations;
Word      iMaxIterations;
/* count of calls to subroutine from main loop */
Word      iSubIterations;
/* Flag to indicate whether work is done */
Word      fWorkDone = FALSE;

char rgbTemplate[] = "!Www! !Nnn! !*d!, !yyyy! !*h!:!0m!:!0s! !2*A!";
Word sTemplate = 45;
char rgbTimeMsg[] = "The system time is:  ";
Word sTimeMsg = 22;
Word iCount;

/* Timer int handler */
void _far HandleTimer()
{
ErcType erc;

    /* if work is not done, indicate timeout.  Else do nothing. */
    if(fWorkDone==FALSE) {
        /* Log the event */
        cTimeOuts++;
        /* notify the client */
        erc = PSend(exchPublic, (Pointer) 0x00000001);
        if(erc != 0)
            Crash(erc);
        }
}

/* Build a timer block, then start the timer. */
Word InitPIT(TPIBType *pTimerblock, Word cPeriods, pFunc pFunction,
             Word fStart)
{
    PointerpData=&exchPublic;

    /* build the TPIB */
    pTimerblock->link1 = 0;
    pTimerblock->link2 = 0;
    pTimerblock->pIntHandler = pFunction;
    pTimerblock->saData = selectorof(pData);
    /* assumes DGROUP for int handler's DS */
    pTimerblock->cIntervals = cPeriods;
    pTimerblock->pRqBlkRet = &pTimerTPIB;
    pTimerblock->footprint = 0;
    pTimerblock->delta = 0;
    pTimerblock->resl = 0;
```

```
    pTimerblock->res2 = 0;

    /* allocate an exchange */
    CheckErc(AllocExch(&exchPublic));

    if(fStart != FALSE)
        CheckErc(SetTimerInt(pTimerblock));

    return(exchPublic);
}

/* Get date and time, then print it.  If we're not behind, delay a */
/* while */
Word DoMyWork()
{

    if(iCount < iSubIterations) {
        PutWord(iCount, 10);
        PutChar(' ');
        iCount++;
        return(FALSE);
        }
    else {
        iCount=0;
        return(TRUE);
        }
}


void main()
{
    Word     iTimeCount;
    Word     exchMyTimer;
    Word     cBytesRet = 0;
    Word     fStartUp = FALSE;
    long     fTimeOut;
    TPIBType  Timerblock;
    pFunc    pTimerIntFunc;
    ErcType  erc=0;

    pTimerIntFunc = &HandleTimer;

    printf("Enter time interval for the PIT (1 = 50 microsec):  ");
    scanf("%d", &iTimeCount);
    printf("\nEnter number of times main loop should execute:  ");
    scanf("%d", &iMaxIterations);
    printf("\nEnter number of times subroutine should be called:  ");
    scanf("%d", &iSubIterations);


    /* set up the Timer */
    exchMyTimer = InitPIT(&Timerblock, iTimeCount, pTimerIntFunc,
                          fStartUp);
    if(exchMyTimer == 0)
        ErrorExit(10);

    /* loop as many times as we want */
    while(iIterations < iMaxIterations) {
        /* start the timer */
        CheckErc(SetTimerInt(&Timerblock));
        putchar('\n');
        PutWord(iIterations, 10);
```

```
        putchar(' ');
        while(TRUE) {
            /* do the work */
            fWorkDone = DoMyWork();
            /* check for a message from the PIT */
            erc = Check(exchMyTimer, &fTimeOut);
            if(erc == 0) {
                WriteBsRecord(bsVid, "\nA timeout occurred", 19,
                              &cBytesRet);
                iCount=0;
                break;
                }
            else if(erc != ercNoMessage)
                CheckErc(erc);
            if(fWorkDone == TRUE) {
            /* if timer went off after we finished, we don't care */
                erc = ResetTimerInt(&Timerblock);
                if(erc == ercOK)
                    break;
                /* if timer went off during timing window */
                else if(erc == ercBadQueue) {
                    printf("\nTimer event in window.");
                    break;
                    }
                else
                    FatalError(erc);
                }
            } /* end of while TRUE */
        iIterations++;
        }

    printf("\nExecution Complete.");
    printf("\nPerformed %d iterations.  %d timeouts occurred.",
            iIterations, cTimeOuts);
```

Chapter 16

The listings in Chapter 16 contain all the actual code in the sample programs. The only parts of the sample programs not shown in that chapter are the #defines of the called procedures.

Chapter 17

Chapter 17 contains no listings.

Chapter 18

Chapter 18 contains the following listings.

SamC.c

This sample program is referenced by Listing 18–1.

```
/*************************************************************************/
/*                                                                       */
/* File Name: SamC.c                                                     */
/* Compiler:  Metaware C                                                 */
/* Date:      07/27/89                                                   */
/* Author:    S. Emmons/A. Coleman                                       */
/*                                                                       */
/*   This program shows use of SAMC calls for asynchronous comm.         */
/*                                                                       */
/*   It uses the following SAMC calls:                                   */
/*            AcquireByteStreamC                                         */
/*            CheckPointBsC                                              */
/*            FillBufferC                                                */
/*            FlushBufferC                                               */
/*            OpenByteStreamC                                            */
/*            ReadByteStreamParameterC                                   */
/*            ReleaseByteStreamC                                         */
/*            SetImageModeC                                              */
/*            WriteByteStreamParameterC                                  */
/*                                                                       */
/*   A sample configuration file, SAMCInt>CommBConfig.Sys is             */
/*   needed for the program and must reside in the                       */
/*   [Sys]<Sys> directory.  The file should be set up with the same      */
/*   parameters used in the InitializePorts() procedure.                 */
/*   A crossed RS-232 cable must                                         */
/*   be attached to communication ports [Comm]A and [Comm]B              */
/*   on the workstation on which the program is to be run.               */
/*                                                                       */
/*   This program must be linked with SamGenAll.obj, or it won't run     */
/*                                                                       */
/*************************************************************************/
#include <stdio.h>
```

```
#include <string.h>
#include <ctype.h>

#define Syslit
#include <CTOSTypes.h>

#define AcquireByteStreamC
#define AllocAreaSL
#define CheckErc
#define CheckPointBsC
#define Exit
#define ExpandAreaSL
#define FillBufferC
#define FlushBufferC
#define NlsULCmpB
#define OpenByteStreamC
#define ReadByteStreamParameterC
#define ReadKbd
#define ReleaseByteStreamC
#define SetImageModeC
#define WriteByteStreamParameterC
#include <CTOSLib.h>

#define FINISH 0x04

/* function definitions */
void SendData(char PortID, Pointer pBSSend, Pointer pBSReceive);
void InitializePorts(void);
void PassData(void);

/*
     See Table 4-3 in the CTOS/VM Reference (Volume 2) for description
     of each field in the Communication Configuration Descriptor.
*/
typedef struct {
     Byte     Type;
     Word     BaudRate;
     Word     StopBits;
     Byte     CharSize;
     Byte     Parity;
     Byte     LineControl;
     Word     Unused;
     Byte     TxMap;
     Byte     RxMap;
     Word     TxTimeOut;
     Word     RxTimeOut;
     FlagType fEOF;
     Byte     bEOF;
     Byte     sTab;
     Byte     cCharsPerLine;
     Byte     sbTxltnFile[79];
     FlagType fNRZI;
     Word     RxBaudRate;
     }    CCDType;

Pointer  pBSCommA;
Pointer  pBSCommB;
Word     cbRet;
Word     cbTQ;
Word     cbRQ;
Word     cbBuffer;
Selector snMemArea;
```

```
Pointer  pbTQA;
Pointer  pbRQA;
Pointer  pBufferArea;
Pointer  pbRet;
CCDType  CommCfgBlk;
char     rgbError[20];
char     rgbMessage[] = "This is a Test Message";

void main () {

ErcType  Erc;
Word     wCompare;
Word     wParam;
Word     i;

/*  Open the byte streams, allocate memory, etc. */
InitializePorts();

/* Set the image mode to binary;  This is not really necessary in this
     example, but if we were transmitting a binary file this would have
     to be done. */
CheckErc (SetImageModeC (pBSCommA, 2));
CheckErc (SetImageModeC (pBSCommB, 2));

/* Use ReadByteStreamParameterC to check one parameter for input BSWA;
     Verify the Line Control for the input BSWA.*/
CheckErc (ReadByteStreamParameterC (pBSCommB, 8, &wParam));
if (wParam != 1)
     printf ("\nLine control for [Comm]B is not set to XON/XOFF");

/* Use WriteByteStreamParameterC to change one parameter for output
     BSWA;Change the tab expansion to zero (treat tab literally) */
CheckErc (WriteByteStreamParameterC (pBSCommA, 12, 0));

/* Use ReadByteStreamParameterC to check one parameter for output BSWA;
     Verify that the tab expansion was changed from 8 to 0.*/
CheckErc (ReadByteStreamParameterC (pBSCommA, 12, &wParam));
if (wParam != 0)
     printf ("\nTab expansion should have been changed to zero.");

/* call my interactive routine */
PassData();

CheckErc (ReleaseByteStreamC (pBSCommB));
CheckErc (ReleaseByteStreamC (pBSCommA));

}    /* End of main () */

void PassData(void) {
char     bPort, bJunk;

while(TRUE) {
     printf("\nEnter sending port letter: ");
     do {
          /* get char */
          CheckErc(ReadKbd(&bPort));
          if(isalpha(bPort))
               putchar(bPort);
          else if(bPort == FINISH)
               Exit(0);
          else bPort=0;
          } while(bPort == 0);
```

```
     /* get carriage return */
     CheckErc(ReadKbd(&bJunk));

     if(toupper(bPort) == 'A') {
          SendData('A', pBSCommA, pBSCommB);
          }
     else if(toupper(bPort) == 'B') {
          SendData('B',pBSCommB, pBSCommA);
          }
     else {
          printf("\nPort must be A or B.  Press any key to Continue.");
          CheckErc(ReadKbd(&bJunk));
          }
     }
}

void SendData(char PortID, Pointer pBSSend, Pointer pBSReceive) {
Pointer pIncomingData;
ErcType Erc;
Byte    bJunk;

CheckErc(FlushBufferC (pBSSend, rgbMessage, strlen(rgbMessage),
                       &cbRet));
/* block till the chars get shipped */
if((Erc = CheckPointBsC (pBSSend, FALSE)) != 0) {
     printf("\nUnable to send data.  Press any key to Continue.");
     sprintf(rgbError, "Erc = %u", Erc);
     printf("\n%s", rgbError);
     CheckErc(ReadKbd(&bJunk));
     }
else if((Erc=FillBufferC(pBSReceive,512,&pIncomingData,&cbRet)) != 0) {
     printf("\nUnable to receive data.  Press any key to Continue.");
     sprintf(rgbError, "Erc = %u", Erc);
     printf("\n%s", rgbError);
     CheckErc(ReadKbd(&bJunk));
     }
else {
     printf("\nSent from Port %c.", PortID);
     *( (char *)pIncomingData + cbRet) = 0; /* terminate string */
     printf("\n%s", pIncomingData);
     }
}

void InitializePorts(void) {
     char rgbPort[40];
     char rgbPortID[40];

/*   Set up the Communication Configuration Descriptor for port A
*/
CommCfgBlk.Type = 3;            /* Type = Communications */
CommCfgBlk.BaudRate = 9600;     /*Transmit at 9600 baud */
CommCfgBlk.StopBits = 1;        /* 1 stop bit */
CommCfgBlk.CharSize = 8;        /* 8 bits per character */
CommCfgBlk.Parity = 0;          /* Parity = None */
CommCfgBlk.LineControl = 1;     /* XON/XOFF */
CommCfgBlk.TxMap = 2;           /* CR/LF */
CommCfgBlk.RxMap = 1;           /* Newline */
CommCfgBlk.TxTimeOut = 5;       /* Transmit timeout = 10 seconds */
CommCfgBlk.RxTimeOut = 5;       /* Receive timeout = 10 seconds */
CommCfgBlk.fEOF = TRUE;         /* bEOF is active */
CommCfgBlk.bEOF = 4;            /* EOF Character */
```

```
CommCfgBlk.sTab = 8;            /* Tab expansion */
CommCfgBlk.cCharsPerLine = 132; /* 132 characters per line */
CommCfgBlk.sbTxltnFile[0] = 0;  /* No translation file */
CommCfgBlk.fNRZI = FALSE;       /* Not XC-002 hardware */
CommCfgBlk.RxBaudRate = 9600;   /* Receive at 9600 baud */

/* Get the port ID for the first port */
printf("Type a port identifier for the first communications port:  ");
gets(rgbPort);
rgbPortID[0]=0;
strcpy(rgbPortID, "[Comm]");
strncat(rgbPortID, rgbPort, 2);

/*   Allocate memory for the Port A BSWA and Queues */
CheckErc (AllocAreaSL(130, &pBSCommA));
snMemArea = selectorof(pBSCommA);
selectorof(pbTQA) = snMemArea;
CheckErc (ExpandAreaSL (1024, snMemArea, &(offsetof(pbTQA)) ));
selectorof(pbRQA) = snMemArea;
CheckErc (ExpandAreaSL (1024, snMemArea, &(offsetof(pbRQA)) ));
cbTQ = (1024 - 12) / 2;
cbRQ = (1024 - 12) / 2;

/*   Open first port using AcquireByteStreamC */
CheckErc  (AcquireByteStreamC  (pBSCommA,  rgbPortID,  7,  modeModify,
&CommCfgBlk,
                                pbTQA, cbTQ, pbRQA, cbRQ, 0, 0));

/*   Allocate memory for the Port B BSWA, and its buffer area. */
selectorof(pBSCommB) = snMemArea;
CheckErc (ExpandAreaSL (130, snMemArea, &(offsetof(pBSCommB)) ));
selectorof(pBufferArea) = snMemArea;
CheckErc (ExpandAreaSL (2048, snMemArea, &(offsetof(pBufferArea)) ));
cbBuffer = 2048;

/* Get the port ID for the second port */
printf("Type a port identifier for the second communications port:  ");
gets(rgbPort);
rgbPortID[0]=0;
strcpy(rgbPortID, "[Comm]");
strncat(rgbPortID, rgbPort, 2);
strcat(rgbPortID, "&[Sys]<Sys>SAMC>CommBConfig.sys");


/*   Open a second port with the specified parameters, using
*    OpenByteStreamC.  NOTE: SAMC>CommBConfig.Sys is identical to the
*    Communication Configuration Descriptor described above, but was
*    created using the Create Configuration File command.  This
*    configuration file must be in [Sys]<Sys>.
*/
CheckErc(OpenByteStreamC(pBSCommB, rgbPortID, strlen(rgbPortID), NULL,
0, modeModify, pBufferArea, cbBuffer));
]
```

FdxIni.c

This sample program module is referenced by Listings 18–2, 18–3, and 18–4.

```
/*
 *  FileName : FDXINI.c
 * Compiler:Metaware C
 * Date: 07/27/89
 * Author:B. Hsueh/A. Coleman
 *           - Bit Synchronous FDX I/O Handler Initialization
 */

#define false    0
#define true     0xFF
#define NIL      ((char *) 0)

/* cases for InitCommLine returned info size */
#define BTOSII2   2
#define BTOS      1
#define CTOS      0

#define  Syslit
#include <CtosTypes.h>
#define  LockOut
#define  LockIn
#define  CheckErc
#define  InitCommLine
#define  ResetCommLine
#include <Ctoslib.h>
#include <interrupts.cf>

#include <comm.h>
#include <fdx.h>


#define  zFlag             0x7e
#define  commWrll          0xb
#define  commWrl0          0xa
#define  commWrl4          0xe
#define  commWrl5          0xf
#define  commlRxEnable     0x10
#define  comml0CrcPreset   0x80
#define  comml0Mark        0x8
#define  commllRxRTxC      0x0
#define  commllTxTRxC      0x9
#define  comml5AbrtIE      0x80
#define  comml5TxUndIE     0x40
#define  comml5CtsIE       0x20
#define  comml5SyncHuntIE  0x10
#define  comml5DcdIE       0x8
#define  commlSpecOnly     0x18
#define  comml_720lRxInt   0x8
#define  commlDma          0xe0
#define  comml4Req         0x4
#define  ercRetArea2Large  67
#define  sFourPortChannel  8

pragma Calling_convention(CTOS_CALLING_CONVENTIONS);
```

```
/** External Functions **/
extern  void  IsrTxData();
extern  void  IsrRxData();
extern  void  IsrExStat();
extern  void  IsrRxSpec();

/** External Variables **/
extern  char     commspec[];
extern  Word     cbcommspec;
extern  char     frxspec;
extern  char     frcvmsg;
extern  Word     iBaudRate;

/** Global Variables **/
FlcbType    *rgpflcb;
ClcbType     commlcb;
InitCommLineRetType   commlret;
Word     wcommlinehandle;
Word     wchiptype;
char         fdoinitcomm;

void        (*pIsrRxData)() = {IsrRxData};
void        (*pIsrExStat)() = {IsrExStat};
void        (*pIsrTxData)() = {IsrTxData};
void        (*pIsrRxSpec)() = {IsrRxSpec};

/** Local Variables **/
static  Word   cbcommlcb;
static  Word   cbcommlret;


void  PrepCommBlk(pflcb,  iSizeRetBlk)
FlcbType  *pflcb;
char       iSizeRetBlk;
{
  cbcommlcb = sClcbTypeMax;
  if (iSizeRetBlk == BTOS)
     cbcommlret = sInitCommLineRetTypeMax;
  else if(iSizeRetBlk == BTOSII2)
     cbcommlret = sBTOSII2RetTypeMax;
  else
     cbcommlret = sInitCommLineRetType;

   commlret.dmahardware = 0;
   commlret.chiptype = 0;
   commlcb.pdsbx    = (char _far *)pflcb;

   commlcb.fdma = false;
   commlcb.ptxisr  = (void _far *)pIsrTxData;
   commlcb.prxisr  = (void _far *)pIsrRxData;
   commlcb.psprxisr =(void _far *) pIsrRxSpec;
   commlcb.pextisr = (void _far *)pIsrExStat;

   commlcb.frawtx    = true;
   commlcb.frawext   = true;
   commlcb.frawrx    = true;
   commlcb.frawsprx = true;
   commlcb.baudtx   = iBaudRate /* 0 */;  /* external clocks */
   commlcb.baudrx   = iBaudRate /* 0 */;  /* external clocks */
   commlcb.fnrzi     = false;

   commlcb.fx2l = false;
```

```
   commlcb.ftdi = false;
   commlcb.ftdixlat= false;
}

/*
-------------------------------------------------------
| NAME  : InitComm

| FUNCTION : Initializes comm i/o
-------------------------------------------------------
*/
char  InitComm(pFlcb)
FlcbType   *pFlcb;
{

Word   commerc ;
char       iSizeRetBlk;

fdoinitcomm = true;
iSizeRetBlk = BTOSII2;

while   (fdoinitcomm) {     /* InitCommLoop */
   if (cbcommspec == sFourPortChannel)
      iSizeRetBlk = BTOS;
   PrepCommBlk (pFlcb, iSizeRetBlk);

   commerc = InitCommLine(&commspec[0],cbcommspec,&commlcb,cbcommlcb,
                          &commlret,cbcommlret);

   if (commerc == ercRetArea2Large)  {
      iSizeRetBlk = BTOS;
      PrepCommBlk (pFlcb, iSizeRetBlk);
      commerc = InitCommLine(&commspec[0],cbcommspec,&commlcb,
                             cbcommlcb,&commlret,cbcommlret);
      }

   if (commerc == ercRetArea2Large)  {
      iSizeRetBlk = CTOS;
      PrepCommBlk (pFlcb, iSizeRetBlk);
      commerc = InitCommLine(&commspec[0],cbcommspec,&commlcb,
                             cbcommlcb, &commlret,cbcommlret);
      }

   CheckErc(commerc);
   fdoinitcomm = false;
   }

return (true);
}

/*
-----------------------------------------
| NAME  : InitFdxIo

| FUNCTION : Initializes comm i/o
-----------------------------------------
*/
void   InitFdxIo(pFlcb)
FlcbType    *pFlcb;
{
```

```
 Word    port;
 char        i;

i = InitComm (pFlcb);

pFlcb->iocommctl = commlret.iocommctl;
pFlcb->iocommdata = commlret.iocommdata;
wcommlinehandle = commlret.commlinehandle;
wchiptype = commlret.chiptype;

port = pFlcb->iocommctl;
pFlcb->fxmitcomplete = false;
frxspec = false;
frcvmsg = false;

rgpflcb = pFlcb;

CLI;
/* set chip to bit-sync mode */
LockOut(port,commWr4);
LockOut(port,comm4Xl+comm4BitSync);
/* set end of frame flag */
LockOut(port,commWr6);
LockOut(port,zFlag);
LockOut(port,commWr7);
LockOut(port,zFlag);
/* enable interrupts. Receive int on first char or special condition */
LockOut(port,commWrl);
LockOut(port,commlRxEnable+commlTxIntEnb+commlExIntEnb+commlStatVect);
/* 8-bit chars, other receive params */
LockOut(port,commWr3);
LockOut(port,comm3Rx8+comm3AutEnb+comm3RxEnb+comm3RxCrcEnb+comm3Hunt);
/* enable int on next receive, set next write to reg. 5 */
LockOut(port,commRxint);
LockOut(port,commRxint+commWr5);
/* set transmit params */
LockOut(port,comm5Dtr+comm5Tx8+comm5CrcCcitt+comm5TxEnb+comm5TxCrcEnb);
/* error reset and set next write to reg 4 */
LockOut(port,commErrRes+commWr4);
LockOut(port,comm4Xl+comm4BitSync);

if (wchiptype == 1) {       /* 82530 only  */
   LockOut(port,commWrl0);
   LockOut(port,comml0CrcPreset); /* not NRZI */
   LockOut(port,commWrl5);
   LockOut(port,
  comml5AbrtIE+comml5TxUndIE+comml5CtsIE+comml5DcdIE+comml5SyncHuntIE);
   LockOut(port,commWrll);
   LockOut(port,commllRxRTxC+commllTxTRxC);
   }
STI;
}
```

Fdxlsr.asm

This sample program module is referenced by Listing 18–5.

```
;-----------------------------------------------------------------
;
;    FILE NAME:    FDXISR.ASM
;    Date:    7/27/89
;    Author:  B. Hsueh
;
;-----------------------------------------------------------------

$EJECT
; ZIN and ZOUT are IN and OUT for Z80-SIO to ensure separation from DMA

%*DEFINE (ZIN(a,b))(
    JMP    $+2
    DB     0F0h, 3Eh
    IN     %a, %b
)

%*DEFINE (ZOUT(a,b))(
    JMP    $+2
    DB     0F0h, 3Eh
    OUT    %a, %b
)

TRUE         EQU 0FFh       ;
FALSE        EQU 00h        ;

commWrl              EQU    1
commWr3              EQU    3
commWr4              EQU    4
commWr5              EQU    5
commRr2              EQU    2
commSabt             EQU    8
commResCh            EQU    018h
commReti             EQU    038h
commErrRes           EQU    030h
commRxInt            EQU    010h
commResTxInt         EQU    028h
commResTxUnd         EQU    0c0h
commResTxCrc         EQU    080h
commResRxCrc         EQU    040h
comm0TxRdy           EQU    4
comm0RxRdy           EQU    1
comm0Dcd             EQU    8
comm0Cts             EQU    020h
comm0TxEom           EQU    040h
comm0Abort           EQU    080h
comm0Sync            EQU    010h
commlRxEof           EQU    80h
commlCrcErr          EQU    040h
commlRxOverrun       EQU    20h
commlReadDma         EQU    060h
commlWriteDma        EQU    040h
commlDmaEnb          EQU    080h
commlExIntEnb        EQU    1
commlTxIntEnb        EQU    2
commlRxIntlst        EQU    8
```

```
comm3Srch           EQU     4
comm3Rx8            EQU     0c0h
comm3RxCrcEnb       EQU     8
comm3Hunt           EQU     010h
comm3AutEnb         EQU     020h
comm3RxEnb          EQU     1
comm4BitSync        EQU     020h
comm5Dtr            EQU     080h
comm5Tx8            EQU     060h
comm5TxEnb          EQU     0Ah
comm5TxCrcEnb       EQU     1
comm5CrcCcitt       EQU     0
commExCtlDcdEnb     EQU     8
commExCtlXmitReq    EQU     4

ercBufferOverrun        EQU     8001
ercLostClear2Send       EQU     8002
ercLostCarrier          EQU     8003
ercOverrun              EQU     8005
ercCrc                  EQU     8217
ercLostData             EQU     8550

; sl0.0
flcbLineErc             EQU     00
flcbExchRet             EQU     02
flcbpXmitBuf            EQU     04  ;flcb.pCurrentXmitBuf
flcbsXmitBuf            EQU     08  ;flcb.sCurrentXmitBuf
flcbiXmitBuf            EQU     10  ;flcb.iCurrentXmitBuf
flcbpRcvData            EQU     16  ;flcb.pRcvData
flcbiRcvData            EQU     20  ;flcb.cbRcvData
flcbcbRcvMax            EQU     22  ;flcb.cbRcvMax
flcbpRecvRgOBuf         EQU     24  ;flcb.pRecvRgObuf
flcbioCommCtl           EQU     32  ;flcb.ioCommCtl
flcbioCommData          EQU     34  ;flcb.ioCommData
flcbfXmitComplete       EQU     36  ;flcb.fXmitComplete

;
; data buffer offsets
;
bufBPrev        EQU     0
bufBNum         EQU     1
bufBNext        EQU     2
bufFInUse       EQU     3
bufSize         EQU     4
bufErcRet       EQU     6
bMsg            EQU     8
bufDataBuf      EQU     9

DGroup          GROUP           DATA
 Data     SEGMENT PUBLIC 'DATA'

 EXTRN rgPFLcb:DWORD

 fRxSpec     DB    1
 fRCVmsg     DB    00h
 ercBuf      DW    0

Data         ENDS

 PUBLIC    IsrRxData,IsrTxData,IsrExStat,IsrRxSpec
 PUBLIC    fRxSpec, fRCVmsg
```

```
FdxIsrCode  SEGMENT  PUBLIC  'CODE'
ASSUME        CS: FdxIsrCode, DS: Dgroup

;---------------------------------------------------------------
;   Receive data
;     routine to handle receive character available interrupt.
;   Input:
;     [DS:BX] - point to FLCB
;   Output:
;     None
;---------------------------------------------------------------

IsrRxData             PROC        FAR
 MOV  DX,WORD PTR[BX+flcbioCommData]
 %ZIN  (AL,DX)                          ; receive a char
 MOV  CX,WORD PTR[BX+flcbiRcvData]      ; check for full buffer
 CMP  CX,WORD PTR[BX+flcbcbRcvMax]
 JGE  IsrRxData_x                       ; if not full
 TEST fRCVmsg, 0ffh                     ; check for end of message
 JNZ  IsrRxData_y
 LES  SI,DWORD PTR[BX+flcbpRcvData]     ; put data in Rcv. buffer
 ADD  SI,CX
 MOV  ES:BYTE PTR[SI],AL

IsrRxData_x:
 INC  WORD PTR[BX+flcbiRcvData]         ;increment buffer position

IsrRxData_y:
 JMP  IsrTxData_0

IsrRxData             ENDP

;---------------------------------------------------------------
; IsrRxSpec:
;    interrupt on parity error, Rx overrun, Frame error or
;    End of Frame.
; Notes:
;   possible LineErc values:
;     ercOverrun, ercCRC, 0
;     ercBufferOverrrun
;        receive EOF and CRC OK, but iRcvData > cbRcvMax
;     ercLostData
;        receive EOF and CRC OK , but no buffer available
;---------------------------------------------------------------

IsrRxSpec             PROC        FAR

 MOV   DX, WORD PTR[BX+flcbIoCommCtl]
 %ZIN  (AL,DX)          ; read status register 0
 TEST  AL, comm0Dcd     ; IF DCD is Lost THEN DO;
 JNZ   IsrRxSpec_2
 MOV   AL,commErrRes    ; In case of CD failure
 %ZOUT (DX,AL)          ;  issue ERROR RESET.
 TEST  WORD PTR[BX+flcbLineErc],0FFFFh
 JZ    IsrRxSpec_1      ;    IF Flcb.LineErc <> 0 THEN
 JMP   IsrRxSpec_w      ;       RETURN

IsrRxSpec_1:            ;    ELSE DO;
 MOV  WORD PTR[BX+flcbLineErc],ercLostCarrier
 JMP  IsrRxSpec_v       ;      Inform The Exchange and RETURN
                        ;    END;
                        ; END;
```

```
IsrRxSpec_2:
 MOV  ercBuf,0          ; Reset temporary variable.
 MOV  AL,1
 %ZOUT  (DX,AL)
 %ZIN  (AL,DX)          ; read status register 1
 MOV   AH,AL
 TEST  AH, commlRxEof+commlRxOverrun
 JNZ  IsrRxSpec_5       ; IF Not (Rx EOF or Rx Overrun) THEN RETURN
 MOV  BYTE PTR[fRxSpec], false
 JMP  IsrRxSpec_y
                        ; ELSE /* Rx EOF or Rx Overrun */
IsrRxSpec_5:
 MOV  AL,commErrRes+commResRxCrc
 %ZOUT  (DX,AL)
 TEST  AH, commlRxOverrun
 JZ  IsrRxSpec_6        ;     IF Rx Overrun THEN DO;

 ;**  clean up the corrupted data
 MOV  DX, WORD Ptr [BX+FlcbIoCommData]  ; get the data port
 %ZIN  (AL,DX)          ;
 %ZIN  (AL,DX)          ;
 %ZIN  (AL,DX)          ;         Read Data Port
 %ZIN  (AL,DX)          ;
 MOV  DX, WORD PTR[BX+flcbIoCommCtl]
                        ;         Set AutoEnable, Hunt Phase,
 MOV  AL,commWr3        ;         8 bits/char, Enable Receiver,
 %ZOUT  (DX,AL)         ;         Set Address Search,
                        ;         Enable CRC Receiver.
 MOV  AL,comm3AutEnb+comm3Hunt+comm3Rx8+comm3RxEnb

IsrRxSpec_5a:
 %ZOUT  (DX,AL)
 MOV  ercBuf,ercOverrun
 JMP  IsrRxSpec_7       ;    END

IsrRxSpec_6:            ;    ELSE DO;
 TEST  AH,commlCrcErr   ;  IF Rx EOF THEN
 JZ    IsrRxSpec_7
 MOV   ercBuf,ercCrc    ; error code

 MOV  DX,WORD PTR[BX+flcbIoCommData]
 %ZIN  (AL,DX)            ; read input data
 JMP  IsrRxSpec_8         ; CRC takes precedence over Overrun
                          ;    END;
IsrRxSpec_7:
 MOV  DX,WORD PTR[BX+flcbIoCommData]
 %ZIN  (AL,DX)            ; read input data
 MOV  AX,WORD PTR[BX+flcbiRcvData]
 CMP  AX,WORD PTR[BX+flcbCbRcvMax]
 JLE  IsrRxSpec_8         ;    IF flcb.iRcvData > flcb.bRcvMax
 MOV  ercBuf,ercBufferOverrun
                          ;    END
IsrRxSpec_8:
 LES  SI,DWORD PTR[BX+flcbpRcvData]
 SUB  SI,bufDataBuf       ; ES:SI Point to the beginning of Buffer
 MOV  AX,ercBuf
 MOV  ES:WORD PTR[SI+bufErcRet],AX
 MOV  AX,WORD PTR[BX+flcbiRcvData]
 DEC  AX
 MOV  ES:WORD PTR[SI+bufSize],AX
 MOV  ES:BYTE PTR[SI+bufFInUse],TRUE
 MOV  BYTE PTR[fRxSpec], TRUE
```

```
; Reinitialize a buffer for next frame reception.
IsrRxSpec_y:
 MOV   WORD PTR[BX+flcbiRcvData],0
 JMP  IsrTxData_0

IsrRxSpec_w:
 MOV   WORD PTR[BX+flcbiRcvData],0
 XOR   AX,AX
 RET

IsrRxSpec_v:
 MOV  WORD PTR[BX+flcbiRcvData],0
 MOV  WORD PTR[BX+flcbiXmitBuf],0FFFFh
 MOV  BYTE PTR[BX+flcbfXmitComplete],TRUE

 MOV  AX,WORD PTR[BX+flcbExchRet]  ;(ax) will have exchange
 MOV  BX,0            ; pMsg = 0
 MOV  DS,BX

 RET
IsrRxSpec            ENDP

;---------------------------------------------------------------------
;
; IsrExStat:
;   Routine to handle interrupt on DCD, CTS transition,
;   Tx EOM, Break/abort
;
;---------------------------------------------------------------------

IsrExStat PROC FAR

 MOV DX, WORD PTR[BX+flcbIoCommCtl]
 %ZIN (AL,DX)           ; read status register 0
 MOV AH,AL              ; save the contents in (ah)
 MOV AL,commRxInt       ; reset the external/status interrupt
 %ZOUT (DX,AL)
 %ZIN (AL,DX)           ; read the status register after reset

 ;** TEST  THE STATUS REGISTER VALUE BEFORE RESET
 TEST AH,comm0Abort     ;
 JZ IsrExStat_Sy        ; If Receive ABORT THEN
 CMP WORD PTR[BX+flcbiRcvData],0 ;
 JE IsrExStat_Sy                 ; Don't abort if no frame recvd
 MOV WORD PTR[BX+flcbiRcvData],0

IsrExStat_Sy:           ;
 TEST AH, comm0Sync
 JZ IsrExStat_1         ; If Receive SYNC THEN
 MOV WORD PTR[BX+flcbiRcvData], 0

 ;** TEST THE STATUS REGISTER VALUE AFTER RESET

IsrExStat_1:
 TEST AL, comm0Dcd      ; IF DCD is Lost THEN DO;
 JNZ IsrExStat_2        ;
 MOV WORD PTR[BX+flcbiRcvData],0
 MOV AL,commErrRes
 %ZOUT (DX,AL)          ; RESET ERROR

 TEST WORD PTR[BX+flcbLineErc],0FFFFh
```

```
 JNZ IsrExStat_w          ;   IF Flcb.LineErc <> 0 THEN RETURN
                          ;   ELSE DO;
 MOV WORD PTR[BX+flcbLineErc],ercLostCarrier
 JMP IsrExStat_y          ;     Inform The Exchange and RETURN
                          ;   END;
                          ; END;
IsrExStat_2:
 TEST AL, comm0Cts        ; IF CTS is Lost THEN DO;
 JNZ  IsrExStat_x
 TEST WORD PTR[BX+flcbLineErc],0FFFFh
 JNZ  IsrExStat_w         ;   IF Flcb.LineErc <> 0 THEN RETURN
                          ;   ELSE DO;
 MOV WORD PTR[BX+flcbLineErc],ercLostClear2Send
 JMP IsrExStat_y          ;     Inform The Exchange and RETURN
                          ;   END;
                          ; END;

IsrExStat_x:
  MOV CX,WORD PTR[BX+flcbiXmitBuf]    ; check for early underrun
  CMP CX,WORD PTR[BX+flcbsXmitBuf]    ; that is, buffer not totaly sent
  JAE IsrExStat_w                     ; if not, do nothing, return
  MOV AL,08                           ; if so,
  %ZOUT (DX,AL)                    ; send an abort then stop transmission
  JMP     IsrExStat_z

IsrExStat_y:
  MOV CX,WORD PTR[BX+flcbiXmitBuf]    ; check for early underrun
  CMP CX,WORD PTR[BX+flcbsXmitBuf]    ; that is, buffer not totaly sent
  JAE IsrExStat_z                     ; if not, stop Xmit, return
  MOV AL,08                           ; if so,
  %ZOUT (DX,AL)                    ; send an abort then stop transmission

IsrExStat_z:
 MOV WORD PTR[BX+flcbiXmitBuf],0FFFFh
 MOV BYTE PTR[BX+flcbfXmitComplete],TRUE
 MOV AX,WORD PTR[BX+flcbExchRet]  ; (ax) will have exchange
 MOV BX,0  ; pMsg = 0
 MOV DS,BX  ;

 RET

IsrExStat_w:
    XOR  AX,AX
    RET
IsrExStat ENDP

;-----------------------------------------------------------------
;   IsrTxData:
;      Routine to handle transmit buffer empty interrupt.
;      It will transmit one character at a time.
;
;   Input:
;        [DS:BX] - point to FLCB
;   Output:
;        None
;-----------------------------------------------------------------

IsrTxData            PROC        FAR

IsrTxData_0:
 MOV   DX, WORD PTR[BX+flcbIoCommCtl]
 %ZIN  (AL,DX)
```

```
 MOV   AH,AL
 TEST  AH,comm0TxRdy     ; If Tx Buffer is Not Empty then
 JZ    IsrTxData_x       ;     RETURN

 MOV  CX, WORD PTR[BX+flcbiXmitBuf]
 CMP  CX, 0FFFFh         ; IF flcb.iXmitBuf=0FFFFh THEN DO;
 JNE  IsrTxData_1        ;     /* No Characters To Send */
 MOV  AL, commResTxInt   ;
 %ZOUT  (DX,AL)          ;     Reset Tx Interrupt
 JMP  IsrTxData_x
                         ; END
IsrTxData_1:
 CMP  CX, WORD PTR[BX+flcbsXmitBuf]
 JNE  IsrTxData_2        ; IF flcb.iXmitBuf=flcb.sXmitBuf
                         ; THEN DO; /* Last Char of Frame */
 MOV  AL,commResTxInt
 %ZOUT  (DX,AL)          ;     Reset Tx Interrupt
 TEST  AH,comm0TxEom     ;     IF CRC has not been sent THEN
 JZ    IsrTxData_x       ;         RETURN
                         ;     ELSE DO;  /* CRC has been sent*/
 MOV  WORD PTR[BX+flcbLineErc],0
 JMP  IsrTxData_y        ;         Inform The Exchange and
                         ;         RETURN
                         ;     END;
                         ; END;
IsrTxData_2:             ; ELSE DO; /* Not Last Character */
 CMP  CX, 1              ; reset Tx Underrun if first char
 JNE  IsrTxData_22
 MOV  AL,commResTxUnd    ;         Reset Tx Underrun
 %ZOUT  (DX,AL)

IsrTxData_22:
 INC  CX                 ;     flcb.iXmitBuf++
 MOV  WORD PTR[BX+flcbiXmitBuf], CX
 MOV  DX,WORD PTR[BX+flcbioCommData]
 LES  SI,DWORD PTR[BX+flcbpXmitBuf]
 ADD  SI,CX
 MOV  AL,ES:BYTE PTR[SI-1]
 %ZOUT  (DX,AL)          ;      Transmit the Data

 MOV  DX,WORD PTR[BX+flcbioCommCtl]
 CMP  CX,WORD PTR[BX+flcbsXmitBuf]
 JNE  IsrTxData_3        ;     IF flcb.iXmitBuf=flcb.sXmitBuf
                         ;     THEN DO; /* Last Character */

 JMP  IsrTxData_x        ;         RETURN
                         ;      END;
IsrTxData_3:             ;      ELSE
JMP   IsrTxData_0        ;          Loop Back
                         ; END;

IsrTxData_x:
 CMP   BYTE PTR[fRxSpec], TRUE
 JE    IsrTxData_w
 XOR   AX,AX
 RET
IsrTxData_w:
 MOV   BYTE PTR[fRCVmsg], TRUE
 JMP IsrTxData_z
 RET

IsrTxData_y:
```

```
 MOV  WORD PTR[BX+flcbiXmitBuf],0FFFFh
 MOV  BYTE PTR[BX+flcbfXmitComplete],TRUE

IsrTxData_z:
 MOV  BYTE PTR[fRxSpec], FALSE

; inform the main process to process the message
 MOV  AX,WORD PTR[BX+flcbExchRet]  ; (ax) will have exchange
 MOV  BX,0    ; pMsg = 0
 MOV  DS,BX    ;

 RET
IsrTxData            ENDP

FdxIsrCode            ENDS
END
```

FdxMain.c

This sample program is not referenced in the text. It is the main module for a synchronous communications program. The program also includes the following modules and include files:

FdxIni.c

FdxParams.c

FdxSub.c

FdxIsr.asm

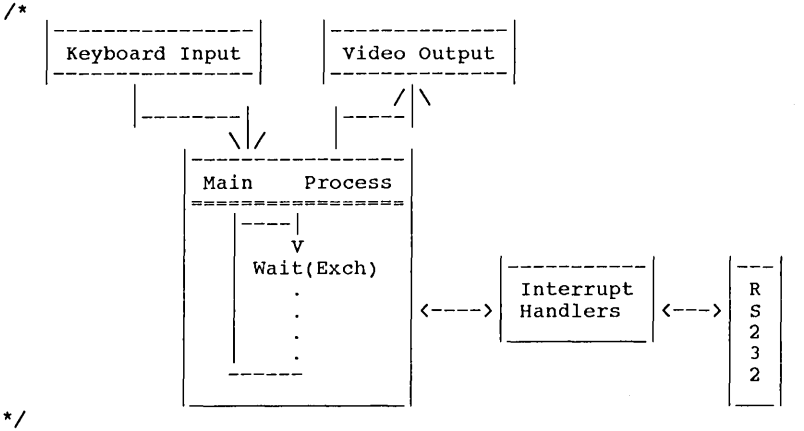
Fdx.h

Comm.h

The interface to the program is documented in the header for this file.

```
/*
 *  FileName : FDXMAIN.c
 *           - Full Duplex Main Loop
 *
 *  Date: 7/21/89
 *  Author: Brian Hsueh, A. Coleman
 *  Compiler: Metaware C 1.0
 *
 *  Inputs from the command line :
 *           [Parameter 1] : communication channel port, A or B.
 *           [Parameter 2] : Data to be transmitted.
 *           [Parameter 3] : Baud rate. Defaults to 0 for Ext Clock
 *
 *  In order to run this program correctly, Following the steps
 *  below.
```

```
 *  1.)  Activate the program by running fdx.run with the proper
 *       parameters.
 *  2.)  After both sides are setup, Hit GO key to synchronize both
 *       processes.
 *  3.)  Hit GO key again to kick off both processes.
 *  4.)  From now on, every time a GO hit, the user data will be
 *       transmitted from one workstation to the other.
 *  5.)  To exit the process, Simply hit FINISH key.
 *
 */
/*
```




```
*/
#include <stdio.h>

#define false    0
#define true     0xFF
#define NIL      ((char *) 0)

#define  Syslit
#include <CtosTypes.h>
#define  CheckErc
#define  Beep
#define  Delay
#define  ErrorExit
#define  Request
#define  ReadKbd
#define  AllocExch
#define  Wait
#include <Ctoslib.h>

#include <comm.h>
#include <fdx.h>

pragma Calling_convention(CTOS_CALLING_CONVENTIONS);

extern  void  InitParams();
extern  void  StartFrameXmission(char  *pbuf, Word  sbuf);

extern char    frcvmsg;
extern FlcbType  *pflcb;
extern char    rcvbuf[];

Word     exch;
KbdType  kbd = {6,0,0,1,0,0,0,53,0,0,0,0,0,0,0,1};
```

```
char    *pmsg;
char     userdata[512];
Word     cbuserdata;
char     keychar;

static   BufType *pbuf = [(BufType *) &rcvbuf[0]];

/*******************************/
/**  Program starts from here  **/
/*******************************/
void  main() [

int   i;

   /**
       Allocate the EXCHANGE for the main process to interact with
       communication interrupt service routines and the keyboard
       inputs.
    **/
  CheckErc(AllocExch(&exch));

  InitParams();  /** see  fdxparams.c  **/

  pflcb->exchret = exch;  /* save the exchange for ISR */
  kbd.pcharret = &keychar;/* initialize the ReadKbd request block */
  kbd.exchresp = exch;

  /**
      Wait until the GO key is depressed to manually synchronize both
      programs simutaneously.
  **/
  printf("\nPress Go to synchronize with remote program.");
  keychar = 00;
  while (keychar != 0x1B) [
    CheckErc(ReadKbd(&keychar));
  ]

  /**
      Kick off the main loop by issuing a read request.
   **/
  printf("\nPress Go to transmit message.\n");
  keychar= 00;
  CheckErc(Request(&kbd));

  /**  MAIN LOOP  **/
  while (true)  [
    CheckErc(Wait(exch, &pmsg));

    if (pmsg != NIL)  [ /** Data from keyboard **/
       if (keychar == 0x1B)  [ /** GO key just hit, start transmitting
          keychar = 0;             data supplied by user  **/
          while (pflcb->fxmitcomplete != true) [
            CheckErc(pflcb->lineerc);
            Delay(1);
            ]
          StartFrameXmission(&userdata[0], cbuserdata);
          ]
       else [
          if (keychar == 0x04) /** FINISH **/
             ErrorExit(0);
          ]
       ]
```

```
    else [ /** process data received from interrupt handler  **/
       if (frcvmsg == true)  [ /** real data from remote side **/
          frcvmsg = false;
          CheckErc(Beep());/** Beep and print out the data received **/
          pbuf->databuf[pbuf->size] = 0;/** add the null terminator **/
          i = puts(&pbuf->databuf[0]);
          ]
       ]

    keychar = 0;  /** Make another read request  **/
    CheckErc(Request(&kbd));
  ]  /** end of main loop  **/

ErrorExit(1);
]
```

FdxParams.c

This program module is not referenced in the text.

```
/*
 *  FileName : FDXPARAMS.c
 * Compiler:Metaware C
 * Date: 07/27/89
 * Author:B. Hsueh
 *           - Full Duplex Parameters Initialization
 *
 */
#define false    0
#define true     0xFF
#define NIL      ((char *) 0)
#include <string.h>

#define  Syslit
#include <CtosTypes.h>
#define  CheckErc
#define  RgParam
#include <Ctoslib.h>
#include <comm.h>
#include <fdx.h>

pragma Calling_convention(CTOS_CALLING_CONVENTIONS);

extern  void  InitFdxIo(FlcbType  *ptr);
extern  char  frcvmsg;
extern  char  userdata[];
extern  Word  cbuserdata;


static  FlcbType    flcb;
        FlcbType   *pflcb = &flcb;
static  BufType    *buf;
        SdType      commchannelsd;
static  char        rgchanA[] = ["[comm]A"};
```

```
static  char        rgchanB[] = {"[comm]B"};
        char        commspec[10];
        Word        cbcommspec;
        SdType     sdBaudRate;
       Word     iBaudRate = 0;

        char        rcvbuf[512];
        char        xmtbuf[512];

void  InitParams() {
Word  bufsize, sxmitbufarea, srecvbufarea, i;
Word  suserdatamax, sxmitbuffmax, srecvbuffmin;

  memcpy(commspec, rgchanB, 7); /** Default to Channel B  **/
  cbcommspec = 7;
  /** Get the Channel Port **/
  if (RgParam(LocCommChan, 0, &commchannelsd) == 0) {
     if ((commchannelsd.pb[0] & 0x5f) == 'A')
        memcpy(commspec, rgchanA, 7);
     else if ((commchannelsd.pb[0] & 0x5f) != 'B') {
             if (commchannelsd.cb <= 2) {
                memcpy(&commspec[6],commchannelsd.pb,commchannelsd.cb);
                cbcommspec = commchannelsd.cb + 6;
                }
             else {
                memcpy(commspec, commchannelsd.pb, commchannelsd.cb);
                cbcommspec = commchannelsd.cb;
                }
          }
      }

  cbuserdata = 256;
  for ( i = 0; i < 256 ; i++ ) {  /* Default Xmit-Data */
  userdata[i] = i;
  }
  /**  Get the User data supplied from parameter 2 **/
  if (RgParam(LocXmtData, 0, &commchannelsd) == 0) {
      memcpy(&userdata[0], commchannelsd.pb, commchannelsd.cb);
      cbuserdata = commchannelsd.cb;
      }

  /**  Get the baud rate from parameter 3 **/
  if( RgParam(LocBaudRate, 0, &sdBaudRate) == 0)
      iBaudRate = atoi( sdBaudRate.pb);

  sxmitbuffmax = 1;
  srecvbuffmin = 1;
  suserdatamax = 259;

  flcb.cbrcvmax = suserdatamax + 4;
  bufsize = suserdatamax + 4 + 1 + 11;

  sxmitbufarea = sxmitbuffmax * bufsize + 2;
  srecvbufarea = srecvbuffmin * bufsize + 2;
```

```
  /**  Initialize the receive buffer pointer  **/
  flcb.precvrgobuf = (char _far *) &rcvbuf[0];
  memset(flcb.precvrgobuf, 0, srecvbufarea);

  buf = (BufType  *) flcb.precvrgobuf;
  buf->bprev = 0;
  buf->bnum = 0;
  buf->bnext = 0;
  buf->finuse = false;

  flcb.prcvdata = (char  _far  *) &buf->databuf[0];
  flcb.cbrcvdata = 0;

  /** Initialize the transmit buffer pointer  **/
  flcb.pxmitrgobuf = xmtbuf;
  memset(flcb.pxmitrgobuf, 0, sxmitbufarea);

  buf = (BufType   *) flcb.pxmitrgobuf;
  buf->bprev = 0;
  buf->bnum = 0;
  buf->bnext = 0;
  buf->finuse = false;

  InitFdxIo(pflcb);
  flcb.fxmitcomplete = true;
  frcvmsg = false;
}
```

FdxSub.c

This program module is not referenced in the text.

```
/*
 *  FileName : FDXSUB.c
 *           - Bit Synchronous FDX I/O handler subroutine
 *  Date: 7/21/89
 *  Author: Brian Hsueh, A. Coleman
 *  Compiler: Metaware C 1.0
 */

#define false    0
#define true     0xFF
#define NIL      ((char *) 0)

#define  Syslit
#include <CtosTypes.h>
#define  LockOut
#define  LockIn
#define  CheckErc
#define  Delay
#include <Ctoslib.h>
#include <interrupts.cf>

#include <comm.h>
#include <fdx.h>
```

```
pragma Calling_convention(CTOS_CALLING_CONVENTIONS);

/**  External Variables  **/
extern  FlcbType   *rgpflcb;

/**
     This routine invokes the transmitter interrupt handler to
     transmit data received from upper user.
 **/
void  StartFrameXmission(pbuffer, cbuffer)
      char     *pbuffer;
      Word      cbuffer;
[
FlcbType    *flcb;
char         stat;

  CLI;     /**  Disable interrupt  **/
  flcb = rgpflcb;
  if ( ~(flcb->fxmitcomplete) == true) {  /** Inconsistency ?? **/
     STI;
     CheckErc(ercBusyBitIo);
     ]

  flcb->pcurrentxmitbuf = pbuffer;
  flcb->scurrentxmitbuf = cbuffer;

  LockOut(flcb->iocommctl, commResTxCrc); /** Reset transmitter CRC **/

  stat = LockIn(flcb->iocommctl); /** Read in comm. port status  **/
  if ((stat & comm0Cts) == 0) {   /** Line disconnected  **/
     flcb->fxmitcomplete = true;
     STI;
     CheckErc(ercLostClear2Send);
     ]
  else {  /** Line is connected normally.  **/
     while((stat & comm0TxRdy) ==0) {/**Check till Txmitter is ready**/
          Delay(1);
           stat = LockIn(flcb->iocommctl);
           ]
     flcb->icurrentxmitbuf = 1;
     LockOut(flcb->iocommdata, *pbuffer);
     flcb->fxmitcomplete = false;
     ]
  STI;
]
```

Fdx.h

This include file is not referenced in the text.

```
/*****************************************************************
 *
 *  File Name  :  fdx.h
 *
 *****************************************************************/
#define  LocCommChan            1
#define  LocXmtData             2
```

```
#define  LocBaudRate  3

/* Internal ercs */
#define  ercBufferOverrun        8001
#define  ercLostClear2Send       8002
#define  ercLostCarrier          8003
#define  ercOverrun              8005
#define  ercCrc                  8217
#define  ercBusyBitIo            8105
#define  ercLostData             8550
#define  ercTimeout              8100

   /* buffer block */
typedef   struct {
          char       bprev;
          char       bnum;
          char       bnext;
          char       finuse;
          Word       size;
          Word       ercret;
          char       bmsg;
          char       databuf[128]}    BufType;

   /* Frame level Line Control Block */
typedef   struct  {
        /* Offset values (0,2,4,8)     */
     Word        lineerc;
     Word        exchret;
     char _far  *pcurrentxmitbuf;
     Word        scurrentxmitbuf;
        /* Offset values (10,12,16,20,22) */
     Word        icurrentxmitbuf;
     char  _far *pcurrentrcvbuffer;
     char  _far *prcvdata;
     Word        cbrcvdata;
     Word        cbrcvmax;
       /* Offset values (24,28) */
     char  _far *precvrgobuf;
     char  _far *pxmitrgobuf;
       /* Offset values ( 32,34,36,37 ) */
     Word        iocommctl;
     Word        iocommdata;
     char        fxmitcomplete;
     char        fill[3]}   FlcbType;

typedef  struct {
         char   scntinfo;
         char   recode;
         char   nreqpbcb;
         char   nresppbcb;
         Word   usernum;
         Word   exchresp;
         Word   ercret;
         Word   rqcode;
         char   reserved[6];
         char _far  *pcharret;
         Word  scharret }  KbdType;

typedef  struct {
         char _far   *pb;
         Word         cb }   SdType;
```

```
typedef  struct {
         char _far     *pdsbx;
         void _far     *ptxisr;
         void _far     *pextisr;
         void _far     *prxisr;
         void _far     *psprxisr;
         char       frawtx;
         char       frawext;
         char       frawrx;
         char       frawsprx;
         Word       baudtx;
         Word       baudrx;
         char       fnrzi;
         char       fx2l;
         char       ftdi;
         char       ftdixlat;
         char       fdma;
         char       fptrxlat;
         char       bunisysreserved;
         char       fproctype; }   ClcbType;

#define  sClcbTypeMax  36
#define  sClcbType     29

typedef  struct  {
         Word  commlinehandle;
         Word  iocommctl;
         Word  iocommdata;
         Word  chiptype;
         Word  pitresolution;
         Word  dmahardware;
         char  fxlatarea;
         char  _far   *pxlatarea;
         Word  cbxlatare;
         Word  iox2l;
         Word  wlinetype}  InitCommLineRetType;

#define  sInitCommLineRetTypeMax  23
#define  sBTOSII2RetTypeMax       19
#define  sInitCommLineRetType     6
```

Comm.h

This include file is not referenced in the text.

```
/* File: Comm.h

   This file defines the common parameter block formats
   for all communications devices
*/

#define dmaClearMask     0x01
#define dmaModeRead      0x49
#define dmaModeWrite     0x45
#define dmaSetMask       0x05
#define dmaStatusComplete   0x02 /*comm channel TC*/
#define ioDmaAdr         0x04
#define ioDmaAdrExt      0x68
```

```
#define ioDmaClearFf    0x18
#define ioDmaMask       0x14
#define ioDmaMode       0x16
#define ioDmaStatus     0x10
#define ioDmaWct        0x06
#define ioIntMask       0x22
#define intClearMask    0xFD
#define intEOI          0x20
#define ioIntCtl        0x20
#define ioCommExtCtl    0x60
#define ioCommIntAck    0x30
#define commWr0         0
#define commWr1         1
#define commWr2         2
#define commWr3         3
#define commWr4         4
#define commWr5         5
#define commWr6         6
#define commWr7         7
#define commRr1         1
#define commRr2         2
#define commNull        0
#define commSabt        1*8
#define commRxint       2*8
#define commResCh       3*8
#define commResExtInt   0x10
#define commEnNxRxInt   4*8
#define commResTxInt    5*8
#define commErrRes      6*8
#define commReti        7*8
#define commResRxCrc    1*64
#define commResTxCrc    2*64
#define commResTxUnd    3*64
#define commVectExtInt  0x1
#define comm0Cts        0x20
#define comm0Dcd        8
#define comm0TxRdy      4
#define comm0IntPnd     2
#define comm0RxRdy      1
#define comm0Hunt       0x10
#define comm0TxEom      0x40
#define comm0TxUndr     0x40
#define comm0Abort      0x80
#define comm0Brk        0x80
#define comm1RxIntlst   8
#define comm1RxEof      0x80
#define comm1RxIntEnb   0x18
#define comm1StatVect   4
#define comm1TxIntEnb   2
#define comm1ExIntEnb   1
#define comm1WriteDma   0x40
#define comm1ReadDma    0x60
#define comm1DmaEnb     0x80
#define comm1Error      0x30
#define comm1FrEnd      0x80
#define comm1CrcErr     0x40
#define comm1ParErr     0x10
#define comm1RxOverrun  0x20
#define comm1FrErr      0x40
#define comm3AutEnb     0x20
#define comm3Rx8        3*64
#define comm3RxEnb      1
```

```
#define comm3Hunt       0x10
#define comm3RxCrcEnb   0x8
#define comm3Srch       0x4
#define comm3SynInh     0x2
#define comm4Syncl      0
#define comm4BitSync    0x20
#define comm4Xl         0
#define comm4Asyl       0x4
#define comm4X16        0x40
#define comm5TxCrcEnb   0x1
#define comm5Rts        0x2
#define comm5Crc16      0x4
#define comm5CrcCcitt   0x0
#define comm5TxEnb      0xa
#define comm5Tx8        3*32
#define comm5Dtr        0x80
#define commExCtlXmitReq    0x4
#define commExCtlXmitDis    0x0
#define commExCtlDcdEnb     0x8
#define commExCtlStd    0x1
#define commExCtlRng    0x4
#define commExCtlDsr    0x2
#define commExCtlSrd    0x1
#define commExStatXmtEnb    0x8000
#define baudRateModeCtr2    0x80
#define baudRateModeCtrl    0x40
#define baudRateMode3       0x36

#define parNone   0x0
#define parEven   0x1
#define parOdd    0x2
```

Chapter 19

Chapter 19 contains the following listings.

GenInt.asm

This sample program is referenced by Listings 19–1 and 19–2.

```
; File Name:   GenInt.asm
; Date:  07/29/89
; Author:Alan Coleman

; This file implements a procedure to generate an software
; pseudo-interrupt.  Also contains interrupt stubs.
;

; Enable use of 386 instructions
$MOD386

extrn TrapHandler1:far
extrn TrapHandler2:far
extrn TrapHandler386:far
extrn PIntHandler1:far

ConstSEGMENT WORD PUBLIC'CONST'
ConstENDS
Data     SEGMENT WORD PUBLIC'DATA'
Data     ENDS
MemorySEGMENT WORD PUBLIC'MEMORY'
MemoryENDS
StackSEGMENT WORD PUBLIC'STACK'
StackENDS

; group all my DGroup stuff
DGroupGROUP Const, Data, Memory, Stack

_CallInt21 segment word 'code'
assume cs: _CallInt21
public CallInt21
CallInt21 proc far

    int 21h
    ret

CallInt21 endp
_CallInt21 ends

_CallInt28 segment word 'code'
assume cs: _CallInt28
public CallInt28
CallInt28 proc far

    int 28h
    ret

CallInt28 endp
_CallInt28 ends
```

```
_asmTrapHandler1 segment word 'code'
assume cs: _asmTrapHandler1
public asmTrapHandler1
asmTrapHandler1 proc far

     mov  bx,sp
     add   WORD PTR ss:[bx],4
     pusha
     call TrapHandler1
     popa
     iret

asmTrapHandler1 endp
_asmTrapHandler1 ends

_asmTrapHandler2 segment word 'code'
assume cs: _asmTrapHandler2
public asmTrapHandler2
asmTrapHandler2 proc far

     pusha
     push ds
     push es
     mov  ax,DGROUP
     mov  ds,ax
     call TrapHandler2
     pop  es
     pop  ds
     popa
     iret

asmTrapHandler2 endp
_asmTrapHandler2 ends


_asmTrapHandler386 segment word 'code'
assume cs: _asmTrapHandler386
public asmTrapHandler386
asmTrapHandler386 proc far

     pusha
     push ds
     push es
     mov  ax,DGROUP
     mov  ds,ax
     call TrapHandler386
     pop  es
     pop  ds
     popa
     ;set an instruction prefix for a 32-bit return
;    db    66h
;    iret
;    do it the 386 way with 12.0 Assembler
     iretd

asmTrapHandler386 endp
_asmTrapHandler386 ends

_asmPIntHandler1 segment word 'code'
assume cs: _asmPIntHandler1
```

```
public asmPIntHandlerl
asmPIntHandlerl proc far

    pusha
    call PIntHandlerl
    popa
    iret

asmPIntHandlerl endp
_asmPIntHandlerl ends

end ; GenInt.asm
```

TestInts.c

This sample program is not referenced in the text. It exercises GenInt.asm.

```
/*************************************************************************/
/*                                                                       */
/* File Name: TestInts.c                                                 */
/* Compiler:  Metaware C                                                 */
/* Date:      07/28/89                                                   */
/* Author:    A. Coleman                                                 */
/*                                                                       */
/* This file installs and attempts to use a number of different          */
/* types of interrupt and exception handler.                             */
/*                                                                       */
/* NOTE:  If you try to use a 386 trap handler on a non-386, a           */
/* truly spectacular crash results.  Set f386 to False on non-386s.      */
/*                                                                       */
/*                                                                       */
/*************************************************************************/

#include <string.h>
#include <stdio.h>

#define Syslit

#define CheckErc
#define ErrorExit
#define nPrint
#define Set386TrapHandler
#define SetDefaultTrapHandler
#define SetIntHandler
#define SetTrapHandler
#include <ctoslib.h>

/* externs */
extern void CallInt21(void);
extern void CallInt28(void);
extern void ReturnFromInt(void);
extern void asmTrapHandler1(void);
extern void asmTrapHandler2(void);
extern void asmTrapHandler386(void);
extern void asmTrapHandler386II(void);
extern void asmPIntHandler1(void);
```

```
/* globals */

PointerpFaultMessage;
char     rgbFaultMessage[128];
FlagType  f386 = FALSE;
PointerpOldTrapHandler;
PointerpJunkTrapHandler;
int      x, y, z;


/* handles divide-by-zero fault */
void TrapHandler1(void) {
     strcpy(pFaultMessage,  "Did  TrapHandler1()  procedure.  Divide  by
0");
     /* set z to non-zero so instruction can restart */
     z=1;
     }

void TrapHandler2(void) {
     strcpy(pFaultMessage, "Did TrapHandler2() procedure");
     nPrint("\nTraps, oh my!!\n", 16);
     }

void TrapHandler386(void) {
     strcpy(pFaultMessage, "Did TrapHandler386() procedure");
     z=1;
     nPrint("\n386 Traps, oh my!!\n", 20);
     }

void PIntHandler1(void) {
     /* we're mediated and DS is set up, so we can use rgbFaultMessage
*/
     strcpy(rgbFaultMessage, "Did PIntHandler1() procedure");
     nPrint("\nInterrupts, oh my!!\n", 21);
     }


main() {

pFaultMessage = rgbFaultMessage;

CheckErc(SetDefaultTrapHandler(0, (Pointer) asmTrapHandler1,
                               &pOldTrapHandler));
CheckErc(SetTrapHandler(0x21, (Pointer) asmTrapHandler2));
CheckErc(SetIntHandler(0x28, (Pointer) asmPIntHandler1,
                       selectorof(pFaultMessage), FALSE, FALSE));

CallInt21();
printf("\n%s", pFaultMessage);
CallInt28();
printf("\n%s", pFaultMessage);

     /* generate a divide-by-zero fault */
y=5;
x = y/z;
z=0;

     /* are we on a 386? */
     /* Set f386 to false if you run this on a non-386 */
f386 = TRUE;
if(f386 == TRUE)
```

```
    CheckErc(Set386TrapHandler(0x21, (Pointer) asmTrapHandler386));

    /* call the various trap handlers or generate faults, etc. */
CallInt21();
printf("\n%s", pFaultMessage);
CallInt28();
printf("\n%s", pFaultMessage);

    /* generate a divide-by-zero fault */
y=5;
x = y/z;

    /* restore the global ones before leaving */
CheckErc(SetDefaultTrapHandler(0, pOldTrapHandler, &pOldTrapHandler));

return(0);
}
```

Chapter 20

Chapter 20 contains no full-program listings.

CTOS open

Programming Practices and Standards

Application Design

PRENTICE HALL
Englewood Cliffs, N.J. 07632