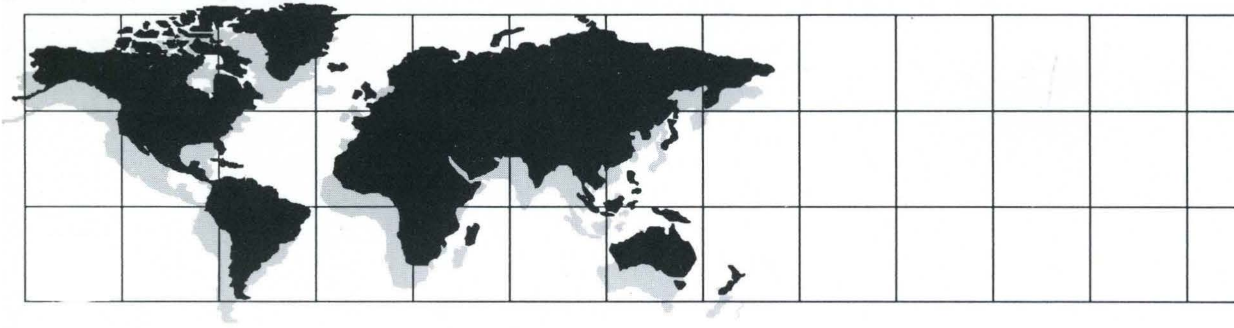


CTOS

Debugger User's Guide

UNISYS

UNISYS

CTOS® Debugger User's Guide

CTOS III 1.0
CTOS II 3.4
CTOS I 3.4
CTOS/XE 3.4
Development Utilities 12.2

August 1992

Printed in USA
4359 4977–000

Priced Item

Page Status

Page	Issue
v through ix	Original
x	Blank
xi	Original
xii	Blank
xiii	Original
xiv	Blank
xv through xxiii	Original
xxiv	Blank
1–1 through 1–4	Original
2–1 through 2–8	Original
3–1 through 3–31	Original
3–32	Blank
4–1 through 4–16	Original
5–1 through 5–12	Original
6–1 through 6–7	Original
6–8	Blank
7–1 through 7–20	Original
8–1 through 8–12	Original
9–1 through 9–4	Original
10–1	Original
10–2	Blank
11–1 through 11–18	Original
A–1 through A–3	Original
A–4	Blank
B–1 through B–6	Original
C–1 through C–8	Original
D–1 through D–51	Original
D–52	Blank
E–1	Original
E–2	Blank
F–1 through F–8	Original
G–1 through G–11	Original
G–12	Blank

Page	**Issue**
H–1 through H–2	Original
I–1 through I–22	Original
J–1	Original
J–2	Blank
Glossary–1 through 7	Original
Glossary–8	Blank
Index–1 through 8	Original

Contents

Figures

Tables

About This Guide

About This Guide

This guide explains how you can use the Debugger as a tool to assist you in identifying and solving software problems. By using the Debugger commands described in this guide, you can

- Examine and change data stored in memory or in registers
- Set and clear unconditional and conditional breakpoints
- Produce displays of memory contents
- Search memory for a pattern of bytes
- Execute program instructions one at a time (single step)
- Disassemble the contents of memory into assembly language source instructions
- Use an application programming interface that allows applications to request debugger services
- Use a second monitor as your debugging screen

For your convenience, the Debugger commands are summarized in the quick reference accompanying this guide.

Purpose and Scope

This guide describes the debuggers for the following operating systems:

- CTOS I, Version 3.4
- CTOS/XE, Version 3.4
- CTOS II, Version 3.4
- CTOS III, Version 1.0

The operating systems are called *CTOS* in this guide. However, when a feature unique to a particular operating system is being described, the specific name of the operating system is provided.

These debuggers are referred to generally as the *Debugger*. (To help you determine which features are supported by your version of the operating system, see Appendix H, "Debugger Features Matrix.")

Who Should Use This Guide

It is assumed that the readers of this guide have some experience with systems or applications programming. Since the instructions displayed by the Debugger are based on the Intel 80X86 family of microprocessors, you should be familiar with these instruction sets although you are not required to understand them in detail. In this guide, you will be introduced to some of the more common instruction sequences that you will encounter again and again in programs you debug.

Furthermore, you can complete the exercises in this guide without knowing standard stack conventions. However, familiarity with the stack and the language compiler you are using is recommended before you can really apply the Debugger commands you will learn in a meaningful way.

A diskette that contains the files you need to work through the exercises in Section 3 and Appendix D is packaged with this guide. For the systems programmer, this guide also provides approaches to debugging crash dump files. These are described in Section 11, "The Executive Command: Debug File."

What is New in This Guide

This edition of the *CTOS® Debugger User's Guide* describes the new Debugger features introduced with CTOS III 1.0. This edition supersedes the CTOS II 3.3 version of the *CTOS Debugger Manual*. To determine which features are supported by your version of the operating system, see Appendix H, "Debugger Features Matrix."

In summary, the new features are

- **CODE-F** has a new optional argument. It allows you to load symbols for a dynamic link library (DLL). See Section 4, "General Purpose Functions and Features."
- The Resource Librarian, a new development utilities tool, adds a resource to a run file. See Section 4, "General Purpose Functions and Features," for information about Debugger run files that include more than one resource.
- **CODE-H** allows you to reissue a previously executed command by history number. See Section 4, "General Purpose Functions and Features."
- **CODE-Y** displays semaphore status. See Section 7, "Display Commands."
- **CODE-Z** displays DLL status. See Section 7, "Display Commands."
- There is now a single application programming interface (API) that allows an application to request debugger services and/or receive notification of debugger events. See Appendix I, "Debugger API."
- With the addition of a VGA graphics controller card, you can attach a second monitor to your computer and use it as your debugging screen. See Appendix J, "Debugging on a Second Monitor."

How This Guide is Organized

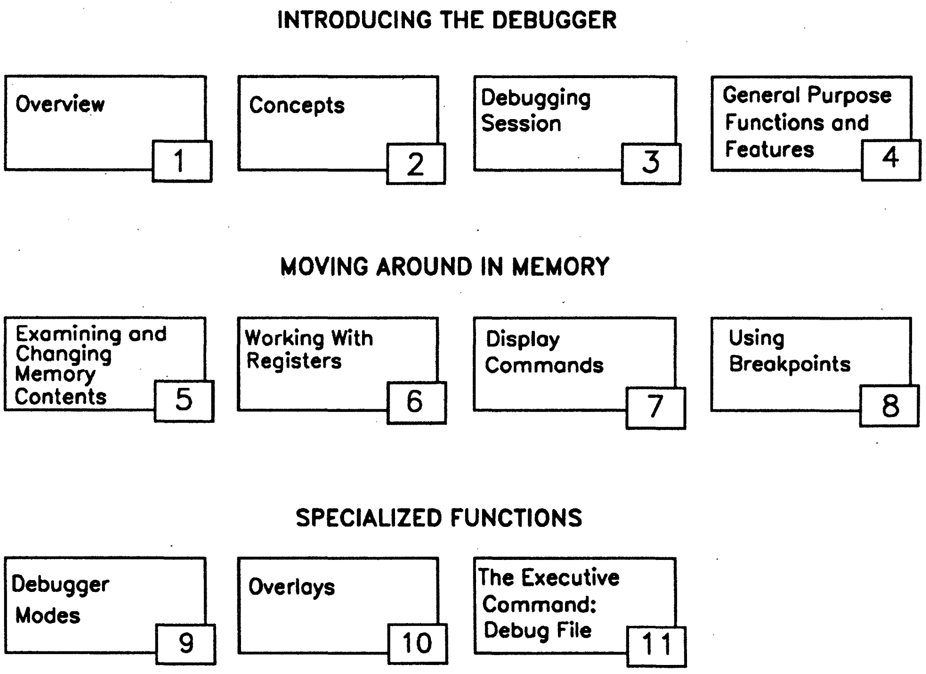
As shown in Figure ATG-1, the sections in this guide are arranged in groups that reflect generalized to specialized debugging activities.

Sections 1 through 4 introduce the Debugger to the first-time user. Included in these sections are a debugging overview, a description of the Debugger concepts such as Debugger command syntax and format, a tutorial which provides hands-on experience with the most commonly used Debugger commands, and a description of the general purpose commands such as those for entering and exiting the Debugger and using the Debugger as a calculator.

Sections 5 through 8 describe the more commonly used debugging commands for examining and moving around in memory. These sections also present different command variations for viewing and changing memory contents. In addition, the sections explain various techniques for setting breakpoints to execute code up to a preset location in your program. Display commands provide you with additional tools you can use for the purposes described.

Sections 9 through 11 are dedicated to specialized debugging commands, such as those for debugging multiprocess programs, overlays, and system crashes.

Figure ATG–1. User's Guide Organization



558.ATG–1

The appendixes augment the sections in this guide by describing

- Status (error) messages
- Shared Resource Processor debugging
- Stack format
- Debugger tips
- Debugger swapping
- Configuration options for using the Debugger
- Extended crash dump process
- Debugger features supported by each operating system version
- Debugger application programming interface
- Debugging on a second monitor

How to Use This Guide

If you are new to debugging, it is recommended that you read the introductory sections. The tutorial in Section 3, "Debugging Session," gives you hands-on experience in using the most common Debugger commands. This should help you understand the command descriptions presented in later sections.

Appendix D, "Debugger Tips," provides additional hands-on exercises using the same tutorial program introduced in Section 3. You are encouraged to try these exercises at any time since they do not assume you have read beyond Section 3.

If you are familiar with general debugging procedures, you can use Sections 4 through 11 of this guide for reference. Although the sections are not totally independent of one another, you are provided with enough cross references to understand any single concept. The quick reference accompanying this guide provides an alphabetized command summary. As a further convenience, the quick reference gives the section numbers where each command is described in detail.

Where to Go For More Information

The *Debugger User's Guide* is one in a group of related manuals describing the operating system and programming subjects. These manuals are described below.

The *CTOS System Administration Guide* describes the administrative aspects of the CTOS II operating system. It includes setting up various operating system types, installing system services, adding application software, and installing peripheral devices. It also describes SRP workstation configuration, system protection, nationalization, and operating system customization.

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

The *CTOS Programming Guide, Volumes I and II* describe techniques for effective programming in the CTOS environment. It concentrates on CTOS specific hardware and system software programming. Volume I covers general topics and Volume II covers extended system services and libraries.

The *CTOS Operating System Concepts Manual* describes the CTOS/XE 3.0/3.3, CTOS I 3.3, CTOS II 3.3, and CTOS III 1.0 operating systems. It provides an explanation of how the operating system works and gives some orientation to the basic concepts the CTOS programmer needs to understand.

The *CTOS Executive User's Guide* describes the most commonly used Executive utilities and features. It provides step-by-step procedures for performing many tasks such as copying files, backing up disks, and creating macros. This guide also includes detailed information about using the file system.

The *CTOS Executive Reference Manual* is organized alphabetically by command name. It includes detailed information about the Standard Software commands and special features of the Executive.

The *CTOS Editor User's Guide* describes how to use the Editor to create or modify an ASCII text file.

The *CTOS Status Codes Reference Manual* contains complete listings of all status codes, bootstrap ROM error codes, and CTOS initialization codes. The codes are listed numerically with any message and an explanation.

The *CTOS Programming Utilities Reference Manual: Building Applications* describes using the Linker, Librarian, and Assembler.

The *CTOS Procedural Interface Reference Manual* covers each of the programming operations for CTOS/XE 3.0/3.3, CTOS I 3.3, CTOS II 3.3, and CTOS III 1.0. It includes the data structures that are available to system services and application programs.

The following Intel manuals introduce the microprocessor instructions for the 80286 and 80386 microprocessors:

- *iAPX 286 Programmer's Reference Manual*
- *80386 Programmer's Reference Manual*
- *i486 Microprocessor Programmer's Reference Manual*

For a tutorial introduction to the basic 8086 instruction set, it is recommended that you examine the 8086/8088 Primer by Stephen P. Morse. You can purchase this manual at your local computer bookstore.

Most Debugger commands work the same in real and protected mode. However, because of addressing differences and the expanded capabilities of protected mode operation, this guide introduces a few commands specifically designed for debugging in protected mode. Additionally, certain existing commands are enhanced.

This guide identifies the features that are unique for protected mode operation in the appropriate sections where they occur. Furthermore, if a particular protected mode concept or structure is introduced, you are directed to an appropriate source for more information.

The following sources provide additional information on protected mode:

- *iAPX 286 Programmer's Reference Manual*
- *80386 Programmer's Reference Manual*
- *i486 Microprocessor Programmer's Reference Manual*
- *CTOS / Open Programming Practices and Standards*
- *CTOS Operating System Concepts Manual*

Conventions Used in This Guide

The following conventions are used throughout this guide.

- In the text, commands that use **CODE** in combination with another key are shown as

  CODE-[the name of the key]

- For readability in **CODE-key** combinations, the name of the key is always shown in uppercase, but use the lowercase letter unless **SHIFT** is specifically indicated. For example:

  CODE-R

  is different from

  CODE-SHIFT-R

- In the Debugger scripts in this guide, the **CODE** commands are shown as

  ^[name of the key]

- Commands that use the arrow keys are shown in the tutorial as

  ↑

  ↓

  →

  ←

In the rest of the guide, the arrow keys are spelled out (i.e., **DOWN ARROW**).

- Memory addresses are indicated as

  addr

  Memory addresses can be segmented addresses (logical or symbolic) or they can be linear addresses. The exceptions are *addr* used with the **CODE-G** and the **CODE-T** commands. In these commands, *addr* can only be a segmented address.

- In the text, **MARK** is spelled out. In the Debugger scripts in the printed version of this guide, **MARK** is also spelled out. For illustrative purposes, there are a few instances in this guide where you will see a solid right triangle. It displays when you press the **MARK** key.

Section 1
Overview

What is a Debugger?

A debugger is a software tool you can use for identifying and correcting run-time errors in your programs. Basically, a debugger eliminates your having to add extra output checking statements to examine the results of execution. With most debuggers, you can set breakpoints to execute portions of your program up to the break you set. Then, you can examine the parameters, for example, that you passed to a procedure, or the results returned after a procedure is executed.

With any debugger, your main concern is to check the code you have written, not the code in system procedures. System procedures have already been tested to work properly for you. Usually, you find errors in the parameters passed to these procedures.

Another common source of error results from incorrect loop logic. You can use a debugger to check the results of looping at various stages of execution.

There are two types of processor modes in which the Debugger operates: real mode and protected mode.

The main reason you would use the Debugger is to check the stack and error codes. You check the parameters passed by examining what gets pushed onto the stack before a call is made. By executing a system call that returns an error code, you can check the error code upon the return. Typically, a zero value indicates successful execution. In this way, you can focus your debugging activity on those procedures that return nonzero values.

It is not within the scope of this guide to provide all approaches you can use to debugging. However, the exercise in Section 3, "Debugging Session," as well as those in Appendix D, "Debugging Tips," should help you with basic approaches to the most common problems you should encounter. In addition, these exercises provide hands-on experience with the most commonly used Debugger commands. The more familiar you are with the commands, the more tools you will have to work with.

CTOS I, CTOS/XE, CTOS II, and CTOS III Debuggers

This guide describes the debuggers for all the following operating systems:

- CTOS I (real mode operating system)
- CTOS/XE (protected mode)
- CTOS II (protected mode)
- CTOS III (protected mode)

Although each of the debuggers is different from the others, with a few noted exceptions, the commands function in the same way whether you use a real mode or protected mode Debugger. For this reason, these debuggers are referred to generally as the *Debugger* in the text of this guide. (To help you determine which features are supported by your version of the operating system, see Appendix H, "Debugger Features Matrix.")

The Debugger runs concurrently with other user and system processes. It responds to commands as you type them. The parameters of Debugger commands can include numeric literals, fundamental 80X86 processor family symbols, and also symbols defined in the program being debugged.

You can use the Debugger with programs written in assembly language, compiled BASIC, FORTRAN, FORTRAN-86, Pascal, PL/M, C, and C++.

Configuration

To use the Debugger, you must configure it in your version of the operating system. (For details, see the *CTOS System Administration Guide*.) The procedure for configuring the Debugger is different if you are using a shared resource processor™ (XE-520 or XE-530) system. (For details, see Appendix B, "Shared Resource Processor Debugging.")

There are, in addition, other Debugger configuration options in the system configuration as well as in the Context Manager™ configuration file. (For details, see Appendix F, "Configuration Options for the Debugger.")

If you attempt to start the Debugger when the Debugger run file is not present, the system beeps. In this guide, it is assumed that your system has a Debugger installed.

Features

The Debugger has several important features:

- It is a symbolic debugger.
- It is a system debugger.
- It supports common debugging techniques.

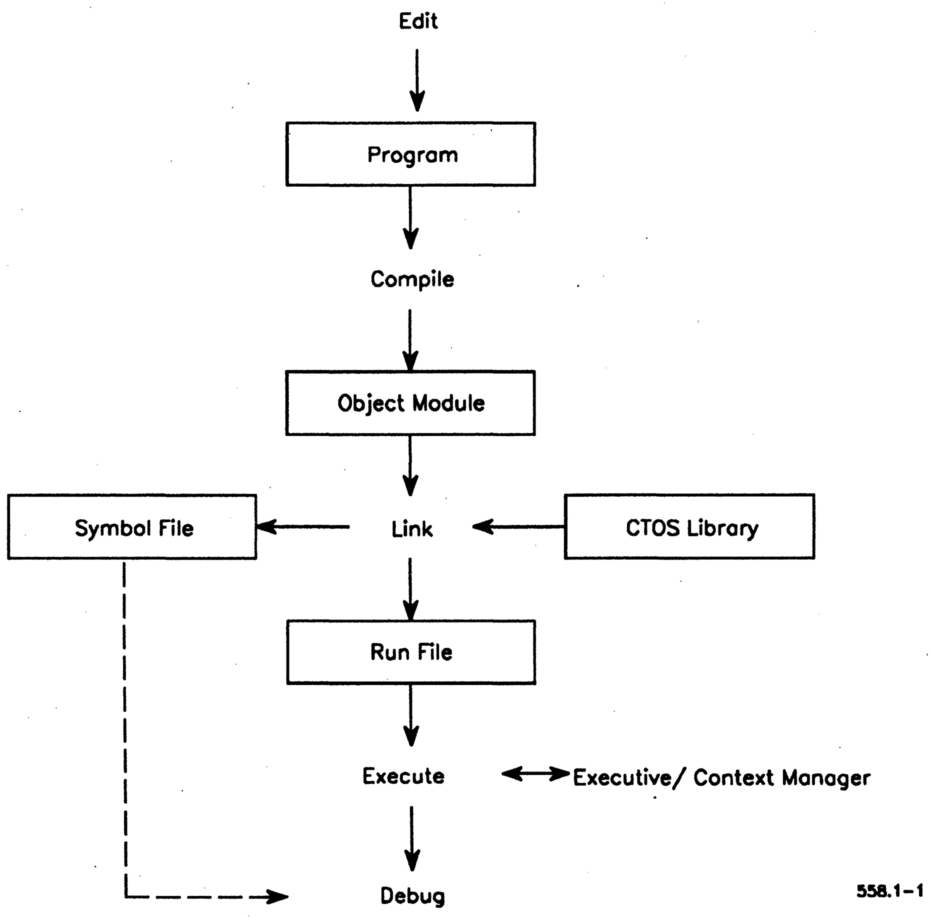
The Debugger is a symbolic debugger. This means that if you have declared an address public in your program and you linked your program such that a symbol file will be generated (as described in the *CTOS Programming Utilities Reference Manual: Building Applications*), you only need to know the symbolic names of the addresses to examine memory, set breakpoints, or perform various functions. To use symbolic names, you load the symbol file for your program. This feature simplifies the debugging process and is described in Sections 3 and 4.

The Debugger is a system debugger as well as an applications debugger. The Debugger is applications-independent, so it does not require that your program be bound to an application. At any time while you are in the Debugger, you can set the Debugger internal process register (PR) to the process you want to debug, whether it be a system or an application process. Then, you can debug that process. Details on how to set PR are contained in Section 6 and later sections in this guide.

The Debugger supports common debugging techniques such as setting and querying breakpoints and executing instructions one at a time (single stepping) through sections of code. Using breakpoints is described in detail in Section 8.

Figure 1–1 illustrates debugging in relation to other steps to developing a program.

Figure 1–1. Debugging Relative to Other Program Development Steps



Section 2
Concepts

This section describes the concepts and terminology that are fundamental to understanding the Debugger functions.

Commands

The Debugger commands invoke various Debugger functions described in detail in this guide. The paragraphs that follow generally describe the command format and command parameters.

Command Format

All Debugger commands have the same format: from 0 to 3 typed-in parameters separated by commas, followed by one command keystroke.

The majority of Debugger commands are entered by holding down the **CODE** key and pressing another key at the same time. Following are examples:

RIGHT ARROW	A command with no parameters
CODE-R	A command with no parameters
38DD (press **CODE-B**)	A command with one parameter
20, DS:10 (press **CODE-D**)	A command with three parameters
112A,2F20, " 'ABC' " (press **CODE-O**)	A command with three parameters

In the preceding examples, **CODE-X** indicates a Debugger command that is given by holding down the **CODE** key while a second key is pressed. Some commands are given by holding down **CODE** and **SHIFT** plus another key. Note that the Debugger is not case sensitive.

Command Parameters

The Debugger accepts parameters similar to the parameters permitted in assembly language. Multiple parameters are separated by commas. The acceptable Debugger parameters are indicated below:

- Constants (numbers, ports, and text).
- Symbolic characters, such as parentheses ( ) or brackets [ ].
- Composite parameters formed using parentheses and separated by commas.
- The minus sign (–).
- The PTR (pointer) operator, which indicates the type of operand you are using, as shown below:

  MOV BYTE PTR [14],2

  MOV WORD PTR [14],2

  In these displays, BYTE PTR points to a byte, and WORD PTR points to a word. If the type of the operand is not implied, you must use the PTR operator. Never use PTR alone, but always with BYTE or WORD.
- The following arithmetic operators:

  + (addition)

  – (subtraction)

  * (multiplication)

  / (division)

- Address expressions, such as those shown below:

  [BX]

  [BP][DI+3] (indexed)

- Symbolic instructions, such as those shown below:

  PUSH BP

  MOV BP,SP

  SUB SP,4

Constants (Numbers, Ports, and Text)

The Debugger recognizes number, port, and text constants. Each of these constant types is described in the paragraphs that follow.

Numbers

The Debugger accepts decimal and hexadecimal numbers. Acceptable numbers are 0 through 9 and A through F. Decimal numbers are indicated by a decimal point (.).

Note: *The Debugger interprets a number that ends with an h, or that does not include either a decimal point or the h, as a hexadecimal number.*

You must include a leading zero (0) preceding the character for a number that begins with the characters A through F. Examples of numbers appear below:

123h	Hexadecimal number
123	Hexadecimal number
123.	Decimal number
0AF	Hexadecimal number
AF	Not a valid number

Ports

A port constant is a number followed by the character i or o. An i indicates that the port is a processor input port. An o indicates that the port is an output port. Examples of port constants appear below:

• 12i	Input port that has port address 12
• 0A2o	Output port that has port address 0A2

The Debugger can both read and write an input port constant. However, the Debugger never reads an output port constant. The **LEFT ARROW** (←) and **RIGHT ARROW** (→) commands open output ports for modification without reading them.

Text

A text constant is a sequence of characters enclosed within single quotation marks. To include a single quotation mark in a text constant, you should type another single quotation mark in front of the mark to be enclosed, for example

'abcd'	The four-character text constant 'abcd'
' 'a'	The two-character text constant 'a (consisting of a single quotation mark and the letter a)

Text constants consisting of one or two characters can be used wherever a number (a numeric constant) can be used. However, text constants consisting of more than two characters can be used only with the **CODE-F**, **CODE-O**, and **HELP** commands.

Symbols

A symbol is a sequence of one or more alphanumeric and special characters. A symbol must begin with an alphabetic character; it cannot begin with a numeral.

The special characters are

- underscore (_)
- period (.)
- dollar sign ($)
- percent sign (%)
- pound sign (#)
- exclamation mark (!)

The Debugger recognizes five types of symbols:

- User-defined public symbols, such as those in a symbol file produced by the Linker from a source program, for example, INIT and CHECKERC. (See the *CTOS Programming Utilities Reference Manual: Building Applications* for details on the Linker.)
- Standard processor register mnemonics, such as AX, BL, and SI. (See "Processor Registers" in Section 6, "Working With Registers.")
- Names of Debugger state variables, such as the process register PR.
- The period (.), which indicates the value of the segment and offset for the most recently opened location. (See "Current Location.")
- The word PatchArea. PatchArea indicates the 50-byte space used for creating conditional breakpoints. (See "Setting Conditional Breakpoints: CODE-A" in Section 8, "Using Breakpoints.")

For examples of symbols, see "Address Expressions."

Symbol Files

A symbol file is used during debugging. The file identifies the locations of public symbols defined in a program so that the Debugger can locate them when you instruct it to do so. The symbol file is created by the Linker and is named

RunFileName.sym

For further details, see Section 4, "General Purpose Functions and Features." Alternatively, symbol information can be added to the run file as a resource. See Section 4, "General Purpose Functions and Features."

Address Expressions

Address expressions in the Debugger have the same structure and semantics as address expressions in assembly language. Examples of address expressions appear below:

sbVerrun	Represents the simplest address expression (a symbol).
RgParam+(100/2)	Is a more complex address expression involving a composite parameter.
[BX+5]	Is an indexed address expression. BX is an index register, and the brackets ([ ]) usually mean the contents are indexed.
ES:[BX+5][SI]	Is a doubly indexed address expression having a segment override prefix. ES is the segment override prefix.

Symbolic Instructions

Symbolic instructions in the Debugger have basically the same structure and semantics as they do in assembly language. If the instruction contains an offset, the Debugger will look up the symbol for it. Examples of symbolic instructions appear below:

```
MOV AX, WORD PTR [BX+5]

LOCK INC [BX]
```

Current Value

The Debugger stores a special value that is either the value most recently displayed by the Debugger, or the value that you typed most recently. This value is called the *current value.*

To display the current value again, type an equals sign (=). You can also display the current value in a different number system. (For details, see "Changing the Base of a Number System: CODE-R" in Section 4, "General Purpose Functions and Features.")

Current Location

The Debugger stores the logical address, that is, the address in the form

```
SA:RA
```

of the most recently opened location. This location is called the *current location.*

Instead of typing out the current location (or its public symbol) as a Debugger command parameter, you can instead refer to it simply by typing a period (.). (For practical examples of how you can use this feature, see Section 3, "Debugging Session." Also see Section 8, "Using Breakpoints.")

Debugger Modes

The Debugger operates in one of three different modes depending on how you've activated it. The modes are *simple, multiprocess,* and *interrupt.* Which mode you use depends upon the mode your program is designed to use. The modes and their associated usages are as follows:

Simple mode	Used for programs that perform a single process user task.
Multiprocess mode	Used for programs that depend on continuous execution of all processes except processes that are explicitly stopped at breakpoints.
Interrupt mode	Automatically invoked when breakpoints are encountered when interrupts are disabled.

The way the Debugger functions in each of these modes is described in detail in Section 9, "Debugger Modes."

Processor Modes

The Debugger also operates in two different processor modes (that is, real mode and protected mode). The mode in which the Debugger is operating is reflected by the Debugger prompt displayed. (For details, see "Debugger Prompts" in Section 4, "General Purpose Functions and Features.")

Section 3
Debugging Session

What to Expect From This Debugging Session

In this section you will use the Debugger to debug a short Pascal program. The program opens a file (byte stream), reads and tests it byte by byte, and writes it to the screen.

This debugging session is intended to acquaint you with some of the more commonly used Debugger commands. Although you are not required to understand assembly language, you will be introduced to some assembly language instruction sequences that you will recognize as patterns again and again in almost every program you debug. In addition, comments accompanying this session describe what is happening on the stack as parameters are pushed onto it and stack space is allocated for local variables. You can complete this exercise without knowing standard stack format, but familiarity with stack conventions is recommended before you do any serious debugging.

Note: *Stack conventions concerning local variables and parameter passing are language specific. This guide shows the stack as it would appear when using either a Pascal or PL/M compiler. For details, see your language manual.*

For your reference, stack conventions are described in detail in Appendix C, "Stack Format." Other useful debugging guidelines, including more practice with this program, are included in Appendix D, "Debugging Tips."

The Program

You will be debugging a Pascal program called DisplayFile. (See Figure 3–1 below.) Pascal is a relatively easy high-level language to read even if you have never coded in it. Following is a brief summary of the main program and procedures.

Note: *In Figure 3–1, the first line displayed (that is, {$Debug–}) is a Pascal compiler metacommand. It is not related in any way to the Debugger commands.*

Main Program

The main program opens a text file (byte stream) in text mode (mt) by calling the OpenByteStream operation. Opening a file in text mode results in the file being read from the beginning up to the formatting information, which is ignored. (Text mode is described in the *CTOS Procedural Interface Reference Manual* in the description of OpenByteStream.)

After called procedures process each byte, the main program closes the file, and the program terminates.

The Process and TypeSub Procedures

To process each byte in the file, the main program calls the Process procedure. Process, in turn, calls TypeSub, which actually performs the processing. TypeSub reads and tests each byte to determine if the byte is text, which should be written to the screen, or if the byte is the end of file (EOF).

When TypeSub detects the EOF, it returns to the Process procedure. Process then writes the string 'Finished' after the last text byte written to the screen.

Figure 3–1. DisplayFile.pas Source Program

```
{$Debug-}
PROGRAM DisplayFile;

CONST
    ercOK = 0;
    ercEOF = 1;
    mt = #6d74;

TYPE

    Pointer = ADS of Byte;
    ErcType = Word;

VAR [EXTERN]
    bsVid: Array [1..130] of Byte;

VAR [PUBLIC]
    BSWA                    :Array [1..130] of Byte;
    done                    :String (8);
    erc                     :Erctype;
    junk                    :Word;
    buffer                  :String (1024);
    count                   :Integer;

PROCEDURE Exit; EXTERN;

FUNCTION OpenByteStream (
    pBSWA                   :Pointer;
    pbFileSpec              :Pointer;
    cbFileSpec              :Word;
    pbPassWord              :Pointer;
    cbPassWord              :Word;
    mode                    :Word;
    pBufferArea             :Pointer;
    sBufferArea             :Word) : ErcType; EXTERN;

FUNCTION CloseByteStream (
    pBSWA                   :Pointer): ErcType; EXTERN;

FUNCTION WriteByte (
    pbs                     :Pointer; b: Byte): ErcType; EXTERN;

FUNCTION ReadByte (
    pbs                     :Pointer;
    pbRet                   :Pointer): ErcType; EXTERN;
```

Figure 3–1. DisplayFile.pas Source Program (cont.)

```
FUNCTION WriteBsRecord (
     pBS                     :Pointer;
     pbmsg                   :Pointer;
     cbs                     :Word;
     pcbret                  :Pointer): ErcType; EXTERN;

PROCEDURE CheckErc (ercCode: Word); EXTERN;

PROCEDURE FatalError (ercCode:Word); EXTERN;

FUNCTION WriteAByte (pbsOut: Pointer;
     b: Byte): ErcType [PUBLIC];

VAR erc : Word;

BEGIN
     WriteAByte := WriteByte (pbsOut, b);
END;


PROCEDURE TypeSub (pbsIn,
     pbsOut: Pointer) [PUBLIC];

VAR b: Byte;
erc : Word;

BEGIN
     WHILE TRUE DO
       BEGIN
         erc:= ReadByte (pbsIn, ADS b);
         If erc = ercEOF then
           RETURN;
         If erc <> ercOK then
         FatalError (erc);
         count := count + 1;
         CheckErc (WriteAByte (pbsOut, b));
       END;
END;

PROCEDURE Process [PUBLIC];
BEGIN
     count := 0;
     done := 'Finished';
     TypeSub (ADS BSWA, ADS bsVid);
     CheckErc (WriteBsRecord (ADS bsVid, ADS done, 9,
        ADS junk));
END;

BEGIN
     CheckErc (OpenByteStream (ADS BSWA,
        ADS 'DisplayFile.pas', 14, ADS NULL, 0, mt,
        ADS buffer, 1024));
     Process;
     Checkerc (CloseByteStream (ADS BSWA));
END.
```

Required Files

The files you need to work through this debugging exercise are on the diskette packaged with this guide. These files are listed and described below:

DisplayFile.pas	The source code for DisplayFile.
DisplayFile.run	The run file for DisplayFile.
DisplayFile.sym	The file to which the Linker wrote a symbol table of the run file when DisplayFile was linked. The symbol table notes the location of all of the public symbols within the program.

You will find these files in the *<Program>* directory on the diskette. Copy these files to your working directory.

Starting the Debugging Session

In the Executive, invoke the Run command. Type **Run** on the command line, and complete the command form as shown below:

```
Run
 Run file         DisplayFile.run
  [Case]          ______________________________
  [Parameter 1]   ______________________________
  [Parameter 2]   ______________________________
  [Parameter 3]   ______________________________
         .
         .
         .
  [Parameter 16]  ______________________________
```

Do not press **GO**. Instead, press the key combination **CODE-GO**. By doing so, the operating system will load the program and enter the Debugger before executing the first instruction.

Note: *It is assumed in this guide that you have the Debugger installed on your workstation. For details, see your operating system release documentation.*

Read the comments accompanying the Debugger script that follows. Note that these comments are numbered to correspond with the script line numbers. You will be asked to type certain characters that appear in boldface in the script. These characters will be introduced in the comments.

Exiting the Debugger

You can exit the Debugger at any time during this session; just press **GO**. However, this program has a bug. If you exit before fixing the bug, the program will terminate with the following message:

```
Application error:  No such file (Error 203)
```

Address Differences

Do not be too concerned if the addresses in the script do not exactly match the results you see on your screen. The values of segment addresses (SAs) are relative to the memory configuration of the workstation upon which this script was created.

Script

Line number	Script
1	`*'DisplayFile.sym'^f`
2	`*DisplayFile^b   GO`
	`*`

Comments

Line number	Comment
1	To load the symbol table file, type the name of the symbol file within single quotation marks, as shown below: `'DisplayFile.sym'` Then press **CODE** and **F** at the same time. (In the text of this guide, pressing a key combination for a Debugger **CODE** command appears as **CODE-KEY**.) In the Debugger, pressing **CODE-F** is displayed as ^f. The script shows these characters in boldface. The comments describe what keys to press to display the boldface characters. In **CODE-KEY** combinations, uppercase is used in this guide for readability. Unless **SHIFT** is specified, use lowercase.
2	To set a breakpoint at the beginning of the program code, type **DisplayFile** (press **CODE-B**) DisplayFile is the starting address of the program code. In Pascal, DisplayFile is equivalent to the procedure ENTGQQ. To execute the program up to DisplayFile (ENTGQQ), you must then press **GO**, but before pressing **GO**, look again at line 2 in the script: `*DisplayFile^b   GO` This line means you are to type the symbol DisplayFile, press **CODE-B**, then press **GO**. The Debugger is case insensitive, so you can type uppercase or lowercase letters. Press **GO**

Script

Line number	Script
3	`Exiting Debugger`
4	`Break at ENTGQQ in process 0D`
5	`*`**`cs:ip MARK`**`  PUSH BP` ↓
6	`*ENTGQQ+1   MOV BP,SP` ↓

Comments

Line number	Comment
3	The Debugger exits and prints the message `Exiting Debugger`
4	When the program has completed execution up to ENTGQQ, control passes back to the Debugger, which displays a description of the break. The description consists of the breakpoint address (ENTGQQ) together with the number of the suspended process (0D). Then the Debugger displays the Debugger prompt (*) and waits for further command input.
5	To display the next instruction to be executed, type **CS:IP** (press **MARK**) CS is the code segment, and IP is the instruction pointer. A triangle appears on the screen when you press **MARK**. Then the Debugger displays (but does not execute) PUSH BP, the next instruction to be executed. PUSH BP saves the value of the base pointer (BP) on the program's stack. Press **DOWN ARROW** (↓) **DOWN ARROW** lets you view (but does not execute) the instruction following PUSH BP.
6	MOV BP,SP sets BP equal to the stack pointer (SP). Press **DOWN ARROW**

Script

Line number	Script
7	*ENTGQQ+3 **MARK** SUB SP,4 ↓
8	*ENTGQQ+7 **MARK** MOV AX, 0F0F8 **RETURN**
9	***BSWA** = 0DBF2:0F0F8
10	***MARK** MOV AX, 0F0F8 ↓

Comments

Line number	Comment
7	SUB SP, 4 subtracts 4 bytes for local or temporary variables. In this case, the main program is allocating 4 bytes for temporary storage. Collectively, this instruction and the instructions on lines 5 and 6 are the stack prologue. This instruction sequence (or a similar sequence, depending upon the program language you are using) occurs at the beginning of the program and at the beginning of every procedure. Press **DOWN ARROW**
8	Line 8 moves the offset of the variable BSWA (first parameter) into AX. Press **RETURN**
9	To show the actual segment address and offset (SA:RA) of the variable BSWA, type **BSWA=** The Debugger displays the address 0DBF2:0F0F8.
10	To redisplay the instruction shown on line 8, press **MARK** The instruction on line 8 is redisplayed because, when you pressed **RETURN** following that instruction, **RETURN** just closed the location but it did not open a new one. Had you pressed **DOWN ARROW**, on the other hand, the next location would have been opened (and a new instruction displayed).

Script

Line number	Script
11	`*ENTGQQ+0A` **MARK** `PUSH DS`
12	`*ENTGQQ+0B` **MARK** `PUSH AX` ↓
13	`*ENTGQQ+0C` **MARK** `MOV AX, 0FD2D` ↓
14	`*ENTGQQ+0F` **MARK** `PUSH DS` ↓
15	`*ENTGQQ+10` **MARK** `PUSH AX` ↓
16	`*ENTGQQ+11` **MARK** `MOV AX,0E` **MOV AX,0F** ↓

Comments

Line number	Comment
11 - 15	To view the instructions on lines 11 through 15, press **DOWN ARROW** The MOV and PUSH instructions on these lines (as well as the MOV instruction on line 8) indicate that a procedure call is about to be made. In this case, the parameters to the next call, OpenByteStream, are being pushed onto the stack. These instructions are described below. On lines 11 and 12, the data segment (DS) and offset of BSWA (first parameter) are being pushed onto the stack. Line 13 moves the offset of the string literal 'DisplayFile.pas' (second parameter) into AX. On lines 14 and 15, the DS and offset are pushed. Press **DOWN ARROW**
16	MOV AX,0E moves the third parameter into AX. The hexadecimal value 0E shown in this instruction is the byte count of the 'DisplayFile.pas' string. If your workstation displays 14. (note the decimal point following 14) instead of 0E, you will need to change the radix (number system base) displayed from decimal to hexadecimal. To do so, just press **CODE-R.** If you count the characters in the 'DisplayFile.pas' string, you will note that there are 15 (0Fh), not 14 (0Eh). If you were to execute this instruction, the program would terminate with error code 203 (No such file). To correct the error, type **MOV AX,0F** to the right of the erroneous instruction, as shown in the script, then press **DOWN ARROW.**

Script

Line number	Script
17	*ENTGQQ+14 **MARK** PUSH AX ↑
18	*ENTGQQ+11 **MARK** MOV AX,0F ↓
19	*ENTGQQ+14 **MARK** PUSH AX ↓
20	*ENTGQQ+15 **MARK** MOV AX, 0FD3C ↓
21	*ENTGQQ+18 **MARK** PUSH DS ↓
22	*ENTGQQ+19 **MARK** PUSH AX ↓
23	*ENTGQQ+1A **MARK** XOR AX,AX ↓
24	*ENTGQQ+1C **MARK** PUSH AX ↓
25	*ENTGQQ+1D **MARK** MOV AX, 6D74 ↓
26	*ENTGQQ+20 **MARK** PUSH AX ↓
27	*ENTGQQ+21 **MARK** MOV AX, 0F136 ↓
28	*ENTGQQ+24 **MARK** PUSH DS ↓
29	*ENTGQQ+25 **MARK** PUSH AX ↓
30	*ENTGQQ+26 **MARK** MOV AX,400 ↓
31	*ENTGQQ+29 **MARK** PUSH AX ↓

Comments

Line number	Comment
17	Line 17 shows the next instruction. To redisplay the previous instruction (the one you corrected), press **UP ARROW**
18	The instruction displayed is correct now on line 18.
19	To move on to the next instruction shown on line 19 again, press **DOWN ARROW**
20 - 31	To view the MOV and PUSH instructions on lines 20 through 31, press **DOWN ARROW** These instructions get the remainder of the parameters to OpenByteStream onto the stack. A few of these are commented on below. On line 23, XOR AX,AX sets the value of AX to 0. On line 24, the fifth parameter (password byte count of 0) is pushed onto the stack. On line 25, the characters 6D and 74 are the ASCII codes for m and t, respectively. Mode text (mt) is the mode in which the file is opened. Line 30 moves the last parameter 400h (1024 decimal) into AX.

Script

Line number	Script
32	*ENTGQQ+2A **MARK** CALL OPENBYTESTREAM ↓
33	*ENTGQQ+2F **MARK** PUSH AX **.^b** **GO**
34	Exiting Debugger
35	Break at ENTGQQ+2F in process 0D
36	***ax** → 0000 **RETURN**

Comments

Line number	Comment
32	This instruction calls OpenByteStream. Figure 3–2 (at the end of this section) shows the stack at this point. Press **DOWN ARROW**
33	PUSH AX is the first instruction to be executed following a return from the OpenByteStream call. The error code from a CTOS call is returned in the AX register. To set a breakpoint at this instruction, type a period (.) (press **CODE-B**) The period (.) means "at the current location." In this case, the location is the address of the instruction PUSH AX. Press **GO**
34	Exiting Debugger
35	Break at ENTGQQ+2F in process 0D
36	To look at the contents of AX, type **ax** (press **RIGHT ARROW**) The Debugger displays 0000. An error code of 0 means that the call to OpenByteStream was successful. Press **RETURN**

Script

Line number	Script
37	`*TypeSub` **MARK** `PUSH BP` ↓
38	`TYPESUB+1` **MARK** `MOV BP, SP` ↓
39	`TYPESUB+3` **MARK** `SUB SP,4` ↓
40	`TYPESUB+7` **MARK** `PUSH WORD PTR [BP+0C]` ↓
41	`TYPESUB+0A` **MARK** `PUSH WORD PTR [BP+0A]` ↓
42	`TYPESUB+0D` **MARK** `LEA AX, WORD PTR [BP-2]` ↓
43	`TYPESUB+10` **MARK** `PUSH DS` ↓
44	`TYPESUB+11` **MARK** `PUSH AX` ↓
45	`TYPESUB+12` **MARK** `CALL READBYTE` ↓

Comments

Line number	Comment
37	To move on to the TypeSub procedure, type **TypeSub** (press **MARK**) PUSH BP starts the stack prologue for the TypeSub procedure. Press **DOWN ARROW**
38	MOV BP, SP continues the stack prologue.
39 - 44	To view the instruction sequence on lines 39 through 44, press **DOWN ARROW** The sequence shows parameters being pushed onto the stack for the call to ReadByte. A few of the lines are described below. Line 39 subtracts 4 bytes from SP to allocate space for local variables. TypeSub has 3 bytes of local variables, but only words (2 bytes each) can be pushed onto the stack. Lines 40 through 44 push the parameters to ReadByte onto the stack. These parameters (pbsIn and ads b) are not public variables but a parameter and an address of a local variable, so their locations are displayed. Square brackets ([ ]) indicate the contents at this location. Line 40 shows the location BP+0C, for example.
45	This instruction calls ReadByte. Press **DOWN ARROW**

Script

Line number	Script
46	TYPESUB+17 **MARK** MOV WORD PTR [BP-4],AX **.^b GO**
47	Exiting Debugger
48	Break at TYPESUB+17 in process 0D
49	***ax** → 0000 **RETURN**
50	***^b**

```
TYPESUB+17
ENTGQQ+2F  count remaining 1
ENTGQQ  count remaining 1
```

Line number	Script
51	***^c** **RETURN**
52	***^b**
53	**WriteAByte** **^b** **GO**
54	Exiting Debugger

Comments

Line number	Comment
46	Set a breakpoint at the return of the call to ReadByte as shown in the script. Then press **GO.**
47	Exiting Debugger
48	Break at TYPESUB+17 in process 0D
49	To examine the value in the AX register, type **ax** (press **RIGHT ARROW**) If there was an error in executing ReadByte, the error code would show up in this register. Press **RETURN**
50	To show all breakpoints set so far, press **CODE-B**
51	To clear all breakpoints, press **CODE-C** Then press **RETURN**
52	Check that the breakpoints are cleared. Press **CODE-B**
53	Set a breakpoint at the procedure WriteAByte as shown in the script. (You do not need to use the period (.) unless you also pressed **MARK** to show the instruction at this location.)
54	Exiting Debugger

Script

Line number	Script
55	`Break at WRITEABYTE in process 0D`
56	**`100.`** **`^p`**
57	`Exiting Debugger`
58	`Break at WRITEABYTE in process 0D`
59	`*`**`count`** → `0065` **`RETURN`**
60	`*^`**`r`**`=101.`
61	`*^`**`r`**`=0065`
62	`*^`**`u`**

Comments

Line number	Comment
55	Break at WRITEABYTE in process 0D
56	WRITEABYTE is called repeatedly in a While loop. With each iteration, one charater from the file *DisplayFile.pas* is written to the screen. To check the loop after 100 iterations, type **100.** (press **CODE-P**) See line 53. Since the Debugger already has breakpointed at WriteAByte once, it will stop executing at the 101st iteration.
57	Exiting Debugger
58	Break at WRITEABYTE in process 0D
59	Examine the value of the Count variable as shown in the script. The Debugger displays 0065, the count in hexadecimal. Press **RETURN**
60	To change this value to 101. (decimal), press **CODE-R** Note the decimal point (.) following the decimal number.
61	Change this value back to hexadecimal by pressing **CODE-R** again.
62	To display the user screen, press **CODE-U** The display shows the output of your program so far, that is, the first 100 characters in the file *DisplayFile.pas*. If you try to count the characters, however, you may run into some difficulty: the count includes all tabs, returns, and spaces.

Script

Line number	Script
63	**`*^c`**
64	**`*^t`**
	`0 F06E WRITEABYTE  (1320,0,0DBF2,0F688,1320)`
	`1 F07C TYPESUB+42  (0DBF2,0F0F8,0DBF2,0F688)`
	`2 F08A PROCESS+29  (282,0EBF6)`
	`3 F094 ENTGQQ+3A`
	`4 F09A BEGXQQ+98`
65	**`*PROCESS+29   ^b   GO`**

Comments

Line number	Comment
63	To return to the Debugger screen, press any key except **CODE-U**. (Once in the Debugger, you can backspace over the character displayed by the key you pressed.) Then, to clear all breakpoints, press **CODE-C**.
64	To display a trace of the stack, press **CODE-T** The Debugger displays the instructions returned to from called procedures. In the script, for example, when WRITEABYTE finishes, it returns to the instruction in the TYPESUB procedure located at TYPESUB+42. Similarly, when TYPESUB finishes, it returns to the instruction at PROCESS+29 in the PROCESS procedure, and so forth. A stack trace displays only the current call sequence. If you were to use **CODE-T** to examine the display again after WRITEABYTE returned to TYPESUB, you would no longer see WRITEABYTE displayed on the first line. Instead, you would see TYPESUB+42. **CODE-T** is useful for quick troubleshooting, because it displays key locations at which you can set breakpoints and check the contents of AX. (For further information, see "Stack Format" in Appendix C, "Stack Format.")
65	To set a breakpoint at the return of the call to TypeSub, type **PROCESS+29** (press **CODE-B**) (You do not need to use the period (.) unless you also pressed **MARK** to show the instruction at this location.) Press **GO**

Script

Line number	Script
66	`Exiting Debugger`
67	`Break at PROCESS+29 in process 0D`
68	**`*cs:ip`** **MARK** `MOV AX,0F688` ↑
69	`PROCESS+24` **MARK** `CALL TYPESUB`
70	**`^c`** **GO**
71	`Exiting Debugger`

Comments

Line number	Comment
66	Exiting Debugger
67	Break at PROCESS+29 in process 0D
68	To display the instruction to be executed at this location, type **cs:ip** (press **MARK**) To verify the call sequence in the stack trace (line 64), there should be a call to TypeSub just before the instruction shown on line 68. To view the previous instruction, press **UP ARROW**
69	This instruction, indeed, is the call to TypeSub.
70	At the end of your debugging session, it is good practice to clear all breakpoints you have set. To do so, press **CODE-C**
71	To exit the Debugger, press **GO** As the program executes to completion, the remainder of the characters in the file *DisplayFile.pas* are output to the screen. Note that the string 'Finished' is displayed following the last byte in the file.

Figure 3–2 shows the stack right after the call to OpenByteStream. Addresses are relative to the workstation's specific memory configuration. See Appendix D, "Debugger Tips," for additional exercises in examining and interpreting what is on the stack.

In the figure, note that the CS and IP are also on the stack, yet there is no visible evidence in the Debugger script (lines 5 through 31) of how they got there. This is because the CS and IP are actually pushed onto the stack by the call to OpenByteStream.

Figure 3–2. Stack After Call to OpenByteStream

Parameters		Stack Contents		
	BP			28E6
				28E4
				28E2
	SP,BP	BP	28E6	28E0
		2 BYTES OF LOCALS		28DE
		2 BYTES OF LOCALS		28DC
pBSWA	SP	S	0DBF2	28DA
		AX	0F0F8	28D8
pbFileSpec		DS	0DBF2	28D6
		AX	0FD2D	28D4
cbFileSpec		AX	F	28D2
pbPassword		DS	0DBF2	28D0
		AX	0FD3C	28CE
cbPassword		AX	0	28CC
mode		AX	6D74	28CA
pBufferArea		DS	0DBF2	28C8
		AX	0F136	28C6
sBufferArea		AX	400	28C4
		CS	8E67	28C2
	SP	IP	2	28C0

Section 4
General Purpose Functions and Features

What are the Debugger General Purpose Functions?

This section introduces basic Debugger functions. They are either basic Debugger commands you use every time you use the Debugger, or they are practical tools that you can use to help you with other Debugging activities. At the most basic level, you certainly need to know how to enter and exit the Debugger. You may need to know why there are several Debugger prompts that can be displayed. Convenient tools help you make calculations or convert a decimal number to its hexadecimal equivalent. Commands specific to debugging programs are described in detail in the remaining sections and appendixes of this guide.

In summary the general purpose functions are:

- Entering the Debugger
- Accessing other shared resource processor boards
- Debugger prompts
- Loading a symbol file
- Exiting the Debugger
- Using the Debugger as a calculator
- Changing the base of a number system
- Deactivating the Debugger

The following general purpose Debugger features are supported on protected mode and real mode CTOS/XE versions of the operating system. These are

- Using the online help file
- Programming the function keys
- Using wild card characters for public symbols
- Invoking Context Manager/VM operations

To determine which of these features is supported by your system, see Appendix H, "Debugger Features Matrix."

Details of General Purpose Functions

General purpose Debugger functions are described in detail in the paragraphs that follow.

Entering the Debugger

You can enter the Debugger in any of the following ways:

- On a workstation, when you press **ACTION** and hold it down while you press the **A** key (a procedure indicated throughout this guide as **ACTION-A**). Executing the Debugger using **ACTION-A** places you in simple mode. In this mode, the Debugger suspends all user processes. On a shared resource processor, you press **HELP-A** instead of **ACTION-A**.
- When you press **CODE-A** in Basic ATE when using the :DebugPort: option. See Appendix G, "Extended Crash Dump Process," for more information.
- When you press **CODE-GO** after typing an Executive command or selecting an application from the Context Manager menu on a workstation.

  This method of invoking the Debugger was used in the tutorial in Section 3, "Debugging Session." It loads the program but does not begin execution until you press **GO**. **CODE-GO** is typically used when you encounter run-time errors in programs you tried to execute. It allows you to examine and correct code before attempting to execute your program.

- When a process reaches a previously placed breakpoint.

  This is an automatic invocation of the Debugger. The mode in which the program was running determines which Debugger mode you will be in when this invocation occurs. (Modes are discussed in "Debugger Prompts.")

- When you press **ACTION-B** on a workstation.

  ACTION-B is the same as **ACTION-A** except that user processes are not suspended. This mode may be useful for debugging real-time multi-process applications. On a shared resource processor, you press **HELP-B** instead of **ACTION-B**.

  The Executive is an example of a multiprocess program. It has two processes: one accepts user input, and the other updates the time. If you were to enter the Debugger using **ACTION-B** (on a character-mapped workstation), you would note that the time display continues to be updated because the Executive clock process is not suspended.

  This method of invoking the Debugger is not commonly used because very few programs are written as multiprocess programs.

- When a process executes an INT 3 instruction.

  This is also an automatic Debugger invocation. In this case your program has executed a Debugger interrupt instruction.

- When a protected mode process executes an instruction that causes a fault. To enter the Debugger automatically on a fault in protected mode, you must specify an option in the operating system configuration file. (For details on all the system configuration file options for the protected mode Debugger, see Appendix F, "Configuration Options for the Debugger.")

The Debugger interprets keystrokes in a way that minimizes accidental invocations of the Debugger or termination of the process being executed. For example, pressing **ACTION** has no effect unless you press one of three other keys (**A**, **B**, or **FINISH**) simultaneously.

Accessing Other Shared Resource Processor Boards: CODE-M

For shared resource processor debugging, there are times when you may want to look at and change the memory contents on a remote board. The remote board may not have a Debugger installed on it, or perhaps the board did not boot and you want to find out why.

To access the memory on a remote board, use the **CODE-M** command from the Debugger prompt. This displays the memory contents of the remote board. Once you have accessed the remote board, you can examine and change the memory contents as well as use all display commands normally. You cannot set breakpoints, however.

To return to the local board, press **CODE-M** again.

CODE-M is supported when you are debugging from real mode to real mode. If you execute the **CODE-M** command in protected mode, you will not be allowed access to a remote board, and an error message is displayed.

Debugger Prompts

Whenever you use the Debugger, the screen shows your most recent dialogue and also shows a Debugger prompt. The type of prompt displayed depends upon three factors: the processor mode, the Debugger mode, or a special condition.

There are three Debugger modes for each processor mode: simple, multiprocess, and interrupt. The two processor modes are real mode and protected mode. The mode in which your program is running determines which Debugger prompt is displayed when the Debugger is invoked.

Two special conditions are indicated by unique Debugger prompts. These conditions are debugging using the Executive command **Debug File** and an abnormal system termination.

For real mode debugging, the Debugger prompts are an asterisk (*), a pound sign (#), a space, a percent sign (%), a greater-than sign (>), or an exclamation point (!).

These prompts are described in detail in subsequent sections in this guide. However, they are briefly described below:

*	The Debugger has suspended the current process.
#	The Debugger has not suspended the current process.
(space)	The Debugger is at an open location.
%	The Executive command **Debug File** is in control.
>	The system has terminated abnormally.
!	The Debugger is at an interrupt level.

You should recognize the asterisk (*) from debugging the DisplayFile program in Section 3, "Debugging Session." The asterisk (*) indicates that the Debugger has suspended execution of DisplayFile, which is a real mode program.

For protected mode, the Debugger prompts are the same as those for real mode but are preceded by either a solid square symbol, indicating protected mode operation (but not virtual 8086 mode), or an empty square, indicating virtual 8086 mode. When the solid square appears preceding the asterisk (*), for example, this means that the Debugger has suspended execution of the current process in protected mode. (*Virtual 8086 mode* is a variant of protected mode incorporating a different addressing style. For details, see the *80386 Programmer's Reference Manual.*)

Symbol Files

The Linker produces symbol files containing the addresses of symbols that you declare public in your program. (For details on the Linker, see the *CTOS Programming Utilities Reference Manual: Building Applications.*) Usually there is only one symbol file produced for each program.

A symbol file is used during debugging. The file identifies the locations of public symbols defined in a program so that the Debugger can locate them when you instruct it to do so. The symbol file is created by the Linker and is named

RunFileName.sym

Symbol files provide a convenience for you when you are debugging. Using a symbol file allows you to enter the public symbol for an address rather than the actual memory address in a Debugger command. If, for example, you defined the Process procedure as a public procedure in your program, the symbol file for your program would allow you to enter the symbol

Process

rather than the logical address of the procedure, such as

0DBF0:0F148

Likewise, the Debugger displays the symbolic names for all addresses in a symbol file. As an example, when using a symbol file, an instruction such as

CALL 0FFEF:336

might appear as

CALL ErrorExit

To take advantage of your program symbol file when debugging, you must open the symbol file first. Otherwise, the Debugger cannot refer to the file.

Special Cases of Opening Symbol Files

Note that under certain circumstances you may need to perform other tasks before you can open a symbol file. If, for example, you are debugging a multiprocess program, you must set the Debugger internal process register (PR) to the process you want to debug first before opening the symbol file. Doing so enables the Debugger to calculate the correct offset at which to load symbols. (For details on setting PR, see Section 6, "Working With Registers.") In cases where you are doing advanced debugging, you may need to calculate the load offset yourself. This may be required, for example, if you are debugging system crash dump files using the Executive command **Debug File**. (For details on calculating the load offset, see Section 11, "The Executive Command: Debug File.")

Note: *Symbols will work for a Global Descriptor Table (GDT) based program as long as the program has been loaded contiguously in the GDT. However, operating systems prior to CTOS III do not guarantee contiguous loading, it is recommended that symbolic debugging of GDT-based programs be performed soon after booting, before the GDT can become fragmented.*

Opening and Closing a Symbol File: CODE-F

Unless you are opening a symbol file under the special circumstances just described, you can open your symbol file simply by typing

'SymbolFileSpec' (press **CODE-F**)

where *SymbolFileSpec* is the specification for the symbol file. The symbol file name must be enclosed within single quotation marks and must include any appropriate path information.

If, for example, the run file name is *Graph.Run* in the directory <Dir>, you can type

'<Dir>Graph.Sym' (press **CODE-F**)

To close the symbol file of the process currently identified by the Process Register, type

' ' (press **CODE-F**)

To load symbols for a dynamic link library (DLL), use the DLL handle with which to associate the symbol file. Type

n, SymbolFileSpec (press **CODE-F**)

where *n* is the DLL handle and *SymbolFileSpec* is the name of the symbol file. For more information on DLL handles, see "Displaying DLL Status: CODE-Z" in Section 7, "Display Commands."

A maximum of 20 symbol files may be open at one time. If you try to open additional symbol files, the Debugger displays the message:

```
Too many open symbol files
```

In this case, you must close a symbol file before you open a new one.

Note: *The Debugger uses the path of the current process (PR) when opening the symbol file.*

It is a good practice to open the symbol file when you begin a debugging session; thereafter, you can use its symbols freely until you end the debugging session.

CODE-F acts as a toggle that enables or suppresses symbolic output. To suppress symbolic output, press **CODE-F**. As a result, the following message appears:

```
Symbols OFF
```

To enable symbolic output again, you press **CODE-F**. When you do so, the following message appears:

```
Symbols ON
```

The **CODE-F** command only suppresses symbolic output: you can use symbols as input to the Debugger any time you are using a symbol file.

Using Symbols as Resources

Instead of separate files, symbols can be part of the run file. In this case, the symbol is called a "resource." When you add a symbol file as a resource, the debugger uses it automatically as you develop your program. This method eliminates the need to load symbol files.

To make a symbol a resource, use the Resource Librarian command. Type **Resource Librarian** on the command line, and complete the command form as shown below:

```
Resource Librarian
  Run or Resource File          App.Run
  [Resources to add]            Debugger.SymbolFile/App.Sym
  [Resources to delete]         ____________________________
  [Resources to extract]        ____________________________
  [Resources config file]       ____________________________
  [Resources list file]         ____________________________
  [Suppress confirmation]       ____________________________
```

In the [Resources to add] field, you can supply just the type and the id for the symbol resource. The type is 20,000 and the id is 1. The field would look like this:

```
  [Resources to add]            20000.1/App.Sym
```

For detailed information about how to use the Resource Librarian, see the discussion on Resource Librarian in the *CTOS Programming Utilities Reference Manual: Building Applications.*

Exiting From the Debugger

To exit from the Debugger,

press **GO**

The Debugger responds by restoring the screen that was present before you entered the Debugger.

After you press **GO** to leave the Debugger, the operating system directs all keyboard input toward a user process.

You can also press **ACTION-FINISH** to terminate the current program and invoke the exit run file, provided that the current program has not called the CTOS procedure DisableActionFinish (TRUE).

Using the Debugger as a Calculator

You can use the Debugger as a calculator at any time. To do so, you press **ACTION-A (HELP-A** on a shared resource processor), enter an expression to be evaluated, and then type an equals sign (=). For example, if you type

3*7=

the Debugger returns 15 (hexadecimal).

You can also use the calculator mode to change the number base in which the Debugger expresses values. For example, if you type

800=

the Debugger returns 800, which simply indicates that the hexadecimal value 800 is equal to itself. To obtain a display of this value in decimal notation,

type **10.** (press **CODE-R**)

The Debugger returns

=2048.

the equivalent of 800h in decimal notation.

For more information about CODE-R, see "Changing the Base of a Number System: CODE-R."

Changing the Base of a Number System: CODE-R

The **CODE-R** command changes the output radix. The *output radix* is the base of a number system in which information is expressed. Decimal, hexadecimal, octal, or any other base from 2 to 16 can be used.

All memory data is displayed using the radix that is in effect at that time. Unless you change it, this radix is hexadecimal.

To set the output radix to another base,

> type *k* (press **CODE-R**)

where *k* is a decimal number from 2 to 16.

To change the output radix back to hexadecimal from any other base,

> type **16.** (press **CODE-R**)

or else simply press **CODE-R**.

If however, the output radix is already hexadecimal, and you type **CODE-R** alone with no parameters, the output radix changes to decimal.

You can also use the **CODE-R** command to display the current value (described in Section 2). To do so, type an equals sign (=) after you specify the base in which you want the values displayed. (In this case, the equal sign specifies the most recent value.)

For example, if you want the hexadecimal value 100 displayed in decimal notation, you would type

> **10.** (press **CODE-R**)

Then you would type

> **100=**

The Debugger responds by displaying the decimal value 256.

Note: *The output radix applies only to numbers that the Debugger displays. Numeric constants that you enter are interpreted or evaluated independently of the output radix. (For details, see "Numbers" in Section 2, "Concepts.")*

Deactivating the Debugger: CODE-K

You use the **CODE-K** command to deactivate the Debugger on CTOS I. (See Appendix H, "Debugger Features Matrix.")

As used in this guide, the term deactivate means to remove a feature from the group of features available to the user. Thus, the **CODE-K** command prevents you from using the Debugger at all until you reboot.

On systems where **CODE-K** is not supported, however, the command is inoperative. If you try to use it, the Debugger displays the message:

```
Deactivation is not necessary in this version of
Debugger
```

When you press **CODE-K** on a real mode system, the Debugger displays a message asking whether you really want to deactivate it. If so, press **CODE-K** again.

Note: *You must deactivate the Debugger using* **CODE-K** *before you can install a different version of the Debugger on CTOS I.*

Details of the Debugger Features

The details of using the following Debugger features are described in the paragraphs that follow:

- Using online help
- Using programmable function keys
- Using wild cards in public symbols
- Invoking Context Manager operations

Using Online Help

The Debugger help file (see your release documentation for the specific help file name) lists all the Debugger commands in alphabetical order and provides the parameters for each. (Note that these commands are also listed and summarized in the quick reference accompanying this guide.) To view the default help file while in the Debugger, press **HELP**.

You can also open any file you have created that you want to use as a help file. To do so,

type '*FileSpec*' (press **HELP**)

where *FileSpec* is the specification (enclosed within single quotation marks) of the file you want to display. You can, for example, open your source file as a help file to view your source code while debugging.

To move through a help file,

press **NEXT-PAGE** or **SCROLL-UP**

If you know that the information you would like to view is at a certain location within a help file, you can skip over to that approximate location by entering one parameter before pressing **HELP** to open the file. To do so,

type *n* (press **HELP**)

where *n* is a number from 0 to 9 (representing 0% to 90%) that indicates approximately how far into the file you will be viewing its contents.

If, for example, you choose to view a command description beginning at approximately the middle of (50% into) the file,

type 5 (press **HELP**)

To reset the help file back to the default,

type '' (press **HELP**)

Resetting to the default file closes any help file you may have opened.

Programmable Function Keys

In addition to the online help file, you can program the function keys (**F2** to **F10**) to record macros. (Note that the function key **F1** is the **REDO** key and cannot be used for this purpose.)

To record a macro in the Debugger, select a function key and press it. Then, enter the keystrokes you want recorded. After you have completed entering keystrokes, press the function key and the code key at the same time. Thereafter, pressing that function key will replay the keystrokes for you.

Reissuing a Command: CODE-H

The **CODE-H** command allows you to reissue a command that you previously typed. When you activate **CODE-H**, it remembers each command you execute by giving it a history number. Then, instead of having to retype a command, just type the history number, and that command is executed again.

To activate, just type **CODE-H**. From that point, all commands you type are given a history number, starting at 1. Then, when you want to reissue one of those commands, type

historynumber (press **CODE-H**)

To toggle the command off, type **CODE-H**.

Using Wild Cards in Public Symbols

As another convenience in debugging, you can use the asterisk (*) and the question mark (?) as wild cards for characters in a public symbol. Wild cards work the same way as they do for characters in a file specification. (For details on wild cards, see the *CTOS Executive Reference Manual.*)

Public symbols are typically used as parameters to Debugger commands. You can use a symbol with wild cards anywhere that you would normally use the symbol. (For details on the format of commands with parameters, see "Commands" in Section 2, "Concepts.") In the Debugger, you type the symbol with wild cards followed by the Debugger command. If the string you type is unique, the Debugger expands the symbol and responds to the command. Typing a symbol that is not unique results in the Debugger displaying the expanded strings for all matching strings.

Unique Symbols

Type a unique symbol such as *ypes* for a public procedure in your program. Following the symbol, enter the Debugger command you want to use (for example, the **RIGHT ARROW** (→) command) as shown below:

type ***ypes*** (press **RIGHT ARROW**)

The Debugger responds by displaying the message

```
Expanding wildcard...
```

followed by the expanded symbolic address, such as TYPESUB. In response to the Debugger **RIGHT ARROW** command, the Debugger displays a single word. A typical Debugger response for this example appears below:

```
TYPESUB 8B55
```

Symbols That Are Not Unique

If the wild card symbol is not unique, the Debugger displays the expanded symbols for all symbols containing the matching string. As an example,

type **readb*** (press **RIGHT ARROW**)

The Debugger responds by displaying all symbols containing the string 'readb' as shown below:

```
Expanding wildcard...

READBYTESTREAMPARAMETERLPT READBYTES READBYTESNOTMMD
READBYTE READBSRECORD ReadByteNotMMD

Expected parameter(s) not found
```

Invoking Context Manager Operations

If you are using Context Manager on a workstation running a protected mode version of the operating system (CTOS II and CTOS III), the Debugger allows you to invoke Context Manager operations. While debugging in the Context Manager, you can use the Context Manager **ACTION** key combinations, such as **ACTION-GO** and **ACTION-NEXT**. You can also switch to a context that is in the Debugger.

This feature provides the convenience of viewing your source code with an editor in one context while debugging your program in another. To switch from the Debugger to your source code, just press **ACTION-NEXT** until you reach the editor context. To get back into the Debugger, press **ACTION-NEXT** until you reach the context being debugged. (See your Context Manager manual for details on the use of **ACTION** keys.)

Note: *You cannot use the* ACTION *keys in this way if you are using* CODE-I *breakpoints, nor can you display the Debugger screen as a windowed context under the Context Manager.*

Section 5
Examining and Changing Memory Contents

Ways to Look at Memory

There are a variety of ways you can examine memory contents when debugging. With the Debugger, you can use pointers to display one or several memory location contents simultaneously. You can also change the contents of a memory location, open a new memory location, or search for specific patterns in memory.

You would typically use memory displays when you have localized the source of an error in your program. If, for example, you suspect there is something wrong with a particular variable, you can use the Debugger and an indirect address to examine the memory location of that instruction. If the contents are correct, you can proceed to the next potential error location. If the contents are not correct or not what you expected, you can either modify the contents or change the instruction.

This section describes the commands you can use to examine and change memory contents. Some examples are provided. For other examples, see the tutorial in Section 3, "Debugging Session," as well as the exercises in Appendix D, "Debugging Tips."

Looking at Memory

You can examine the contents of memory by typing a parameter that designates a machine address, a register, or an internal Debugger register, followed by a command (**LEFT ARROW**, **RIGHT ARROW**, **BOUND**, or **MARK**). The Debugger displays the contents of the designated address or register and opens that address or register so that you can change its contents. The commands are as follows:

- To display a single byte,

  type *addr* (press **LEFT ARROW**)
- To display a single word,

  type *addr* (press **RIGHT ARROW**)
- To display a double word value,

  type *addr* (press **BOUND**)
- To display a symbolic instruction,

  type *addr* (press **MARK**)

For example, to display 1 byte at address 0A1, you type

0A1 (press **LEFT ARROW**)

The Debugger will return 1 byte of data, such as

1F

To display one word starting at location DS:100, you type

DS:100 (press **RIGHT ARROW**)

The Debugger returns one word of data, such as

1F20

To display the instruction at CreateISAM+10, you type

CreateISAM+10 (press **MARK**)

The Debugger returns the instruction at the symbolic address CreateISAM+10. For example, the **MARK** command might return the instruction

MOV AX, WORD PTR [BX]

Notice that when you press **MARK**, a small right triangle appears on the screen to the right of what you typed. The triangle looks like this:

◣

Using Pointers to Display Memory Contents

You can also use indirect addresses to examine bytes, words, and instructions. You do so by specifying the address of a long pointer that addresses the byte, word, double word, or instruction that you want to examine.

To display a single byte that is addressed by a long pointer,

> type *addr* of byte pointer (press **CODE-LEFT ARROW**)

To display a single word that is addressed by a long pointer,

> type *addr* of word pointer (press **CODE-RIGHT ARROW**)

To display a symbolic instruction that is addressed by a long pointer,

> type *addr* of instruction pointer (press **CODE-MARK**)

After you enter the **CODE-LEFT ARROW, CODE-RIGHT ARROW, BOUND**, or **CODE-MARK** command, the open location is the location addressed by the pointer.

For example, suppose you want to display the byte addressed by a pointer at location 244. You could first fetch the pointer and then fetch the byte, as shown by the following sequence:

> Type **244** (press **RIGHT ARROW**) 04AE
>
> Type **246** (press **RIGHT ARROW**) 2199
>
> Type **2199:04AE** (press **LEFT ARROW**) 00

or you could simply use the **CODE-LEFT ARROW** command as shown below:

> Type **244** (press **CODE-LEFT ARROW**) 00

Displaying Several Locations at Once

You can also use the memory examination commands to display the memory contents of several locations at once. For example, you might want to look at a vector table, parameter list, or symbol table.

To do so, precede the parameter you type with a number indicating how many locations you want displayed. In response, the Debugger displays the specified number of parameters and keeps the last parameter open.

For example, to display three words starting with the word that begins at location DS:100, you type

3, DS:100 (press **RIGHT ARROW**)

A typical Debugger response appears below:

```
 3, DS:100  →  1F20
 0AF:102    →  2F30
 0AF:104    →  30FA
```

Opening and Closing Memory Locations

The memory location that is open changes with each command that modifies the contents of memory. There are three such commands:

RETURN

UP ARROW

DOWN ARROW

Pressing **RETURN** closes the previously open location and does not open any new locations.

Pressing **UP ARROW** opens the previous location.

Pressing **DOWN ARROW** opens the next location.

The Debugger interprets the words *next* and *previous* according to the type of location that is open. *Next* can refer to the next byte, the next word, or the next instruction. If, for example, a word is displayed at the open location, pressing **DOWN ARROW** opens the next location and displays a word at that location also.

Understanding the Debugger Prompts

The Debugger always prompts you when it is ready for more input. The type of prompt depends on whether the location is open, as explained below:

- If no location is open (which happens if the Debugger was just entered, or if **RETURN** was pressed, closing all open locations), the Debugger issues the appropriate prompt. (For details on the types of prompt, see "Debugger Prompts" in Section 4, "General Purpose Functions and Features.")
- If a location is open, the Debugger prompts you with an empty space. This prompt appears on the same line as the value of the open location.

Displaying the Contents of Memory: CODE-D

The **CODE-D** command displays the contents of memory both numerically and in ASCII or, optionally, in EBCDIC. **CODE-D** can accept two or three parameters and can display a number of bytes of memory, along with ASCII equivalents of those bytes, limited only by the amount of memory present in the machine. The Debugger displays the memory contents in columns.

CODE-D with Two Parameters

Most of the time, you will use the two-parameter form of **CODE-D**. In this form, the command is

k, addr (press **CODE-D**)

where *k* is the number of bytes, and *addr* is the memory location. This form causes the Debugger to display a specific number of bytes of memory starting at a specific memory location.

This command is particularly useful in verifying the results of moving data from one location to another. In Appendix D, "Debugger Tips," there is an exercise on moving bytes in a string. A portion of that string move that makes use of the **CODE-D** command is discussed next. (For details, see "String Moves" in Appendix D, "Debugger Tips.")

In the exercise, you move two bytes of the string 'Finished' to the destination, which starts at the symbolic address done. After you execute one MOVSW (move word) instruction, you use **CODE-D** to display the results. To use **CODE-D**, you type

8, done (press **CODE-D**)

As a result, the Debugger displays 8 bytes of memory starting at the logical address of done, as shown below:

```
0DBF2:0F12A   46 69 00 00 00 00 00 00 00  Fi
```

Note that the first two characters in the string 'Finished' are displayed in ASCII to the right of the memory contents.

If you wanted to display the same 8 bytes of memory in EBCDIC, you would use **CODE-SHIFT-D** instead of **CODE-D**.

CODE-D with Three Parameters

A variation of the **CODE-D** command permits you to specify the length of each line you want to display.

When three parameters are specified, the form of the command is

k, l, addr (press **CODE-D**)

where ***k*** is the number of lines, ***l*** is the line length with a default value of 16, and *addr* is the address. If *addr* is set to 0, the current address is used.

For example, to display five lines of memory, 4 bytes long, beginning at the current address, you type

5, 4, 0 (press **CODE-D**) (or press **CODE-SHIFT-D** for EBCDIC)

A typical Debugger response is shown below:

```
0:0     F1   03   80   00   .... [ASCII version]
0:4     10   00   7F   00   ....
0:8     20   00   7E   00   ....
0:0C    30   00   7D   00   ....
0:10    40   00   7C   00   ....
```

The Debugger automatically turns off the symbolic display of addresses while memory contents are being displayed using **CODE-D**.

Changing the Contents of a Memory Location

During a debugging session, you can change the contents of a memory location. For example, suppose you want to change the contents of DS:101 from 2F30 to 2F37. Location DS:101 is open, and the Debugger prompts you with a space.

Press **RIGHT ARROW** (→) as shown below:

```
DS:101 → 2F30
```

Then type

2F37

followed by a **RETURN** to close the location, or by either **UP ARROW** or **DOWN ARROW** to close the location and open a new location.

The contents of location DS:101 are corrected to contain 2F37.

If you corrected the bug introduced in the debugging session in Section 3, you have already tried this feature. The comments describe a bug you encounter when you try to open the file *DisplayFile.pas*. The erroneous instruction located at ENTGQQ+11 in that program is shown again below:

```
MOV AX,0E
```

You realize that you should have moved 15 (0Fh) bytes instead of 14 (0Eh). To correct the contents at this location, you entered the correct instruction in its entirety, that is, you typed

```
MOV AX,0F
```

at the open location. (If you have not already done so, you may want to examine this correction in the context of the tutorial in Section 3.)

Note that in the preceding example, the instruction itself did not actually change to a different one: just the value of the parameter changed. The subject of changing instructions of varying lengths requires special attention and is described next in this section.

Changing Instructions

Assembly language instructions can vary in length from one to several bytes. Therefore, it is important that you adjust for variances when you change instructions in the Debugger.

If, for example, you use a Debugger command to overwrite an instruction with one that is shorter in length, the excess bytes of the original instruction are left as garbage in memory following the new instruction unless they are accounted for in some way. In such a case, you need to replace each extra byte of the original instruction with a No Operation (NOP) instruction. The NOP instruction is a 1-byte instruction that acts as a place-holder.

For instance, suppose you want to overwrite a 3-byte comparison instruction with a 2-byte jump instruction. To do so, you must insert a NOP instruction after the jump instruction, as shown in the example below:

```
24D:120   CMP AX, WORD PTR[BP+4] JMP +2

24D:122   ADD AL.75 NOP
```

The NOP instruction ensures that the last byte of the comparison instruction is not left dangling in memory.

Searching for a Byte Pattern in Memory: CODE-O

The **CODE-O** command searches for a byte pattern in memory. A *byte* pattern is a user-defined group of byte specifiers separated by commas and enclosed in double quotation marks. A *byte specifier* is either a sequence of 2-digit hexadecimal numbers, or a string of characters enclosed in single quotation marks. Examples of byte patterns appear in Table 5–1. Nonliteral byte patterns are enclosed in double quotation marks. Literal byte patterns are enclosed in single quotation marks within the double quotation marks, as illustrated in the table.

The **CODE-O** command can be used, for example, to search for a matching element in an array or to find the next comma in a character string.

To search for a given byte pattern,

type *lower addr, upper addr,* "byte pattern" (press **CODE-O**)

Table 5–1. Examples of Byte Patterns

Byte Pattern	Pattern Specified
"31,32,33"	123
"'ABC'"	ABC
"41,42,43"	ABC
"31,32,33,'ABC',34,35,36"	123ABC456

For example, to search for the byte pattern 31,32 within the range of addresses from 5FE6:0C to 5FE6:100, type

5FE6:0C, 5FE6:100, "31,32" (press **CODE-O**)

The Debugger searches for a byte pattern within the range of addresses starting at *lower addr* and ending at *upper addr*. If the pattern is found, the Debugger displays the pattern at the address at which it was found and changes *lower addr* to the address of the first byte following the pattern.

To make the Debugger continue the search beginning at the new *lower addr*, press **CODE-O** with no parameters.

If the Debugger does not find a byte pattern, the Debugger ends the search when it reaches *upper addr*.

Displaying Physical Addresses: CODE-=

On protected mode systems that employ paging, you can use the **CODE-=** command to display the physical address of a logical or linear address. (For details on paging, see "Introduction to Protected Mode" in the *CTOS Programming Guide, Volume I.*) To display the physical address of *addr*,

type *addr* (press **CODE-=**)

For example, if you typed a logical address, such as 0EF33:6A, and pressed **CODE-=** as shown below

0EF33:6A (press **CODE-=**)

the Debugger responds with the corresponding physical address, such as

000EF9A

Reading and Writing to Ports

To read and write to ports, use **LEFT ARROW** and **RIGHT ARROW** commands together with constants. For example, to read from the byte-input port 17i, type

17i (press **LEFT ARROW**)

To read from the word-input port 31i, you type

31i (press **RIGHT ARROW**)

In either of these cases, after you type the constant, the port becomes an open location. You can then specify a new parameter to be written to the port. Unlike reading from memory, reading from a port can change the state of the system. For example, reading a character from the keyboard removes that character from the keyboard.

Assigning Names to Addresses: CODE-[

You can use the **CODE-[** command to assign a name to the current open address. To do this, type

'AddressName' **CODE-[**

where *AddressName* is the assigned name. This name must be enclosed within single quotation marks. In subsequent commands, you can refer to the address by supplying an ampersand followed by the assigned address name. For example, to display the instruction at an address with the name pcbRet, you can type

&pcbRet (press **MARK**)

You can also use the **CODE-[** command to assign a name to the value at the current open address. To do so, type

n, 'ContentsName' **CODE-[**

where *n* is the size (in bytes) of the value at the current address, and *ContentsName* is the name assigned to that value. The value *n* may be 1, 2, or 4.

To display the number represented by a name, type

&Name=

where *Name* is the name assigned to either an address or the value at an address. For example, if *Name* represents the address 23B4, the Debugger responds by displaying the following value:

23B4

The **CODE-[** command is used by script files to access memory locations and their contents. When a script file is active, the Debugger reads commands from this file rather than the keyboard. Commands in a script file are predefined, but the memory addresses of data items may vary with each debugging session. However, the **CODE-[** command allows a script file to operate regardless of the location of data items.

As an example, suppose the script file must access a particular field in a data structure. Initially, the script file examines the contents of a memory address to obtain a pointer to this field. However, the pointer to the field may be different for each debugging session. Since commands in the script file are predefined, they cannot directly refer to the address where the field is currently located.

In this case, the **CODE-[** command could be used to assign a name to the memory address of the field. Subsequent script file commands would always work because they would always use the assigned name.

Section 6
Working With Registers

This section describes the Debugger's internal process register and explains how to examine and modify registers.

The Process Register

The process register (PR) is a Debugger internal register as opposed to being a processor register (such as AX or DS). PR always identifies and keeps track of the current process. The *current process* is either the process that most recently reached a breakpoint or the process that owns the keyboard when you enter the Debugger.

PR is automatically set to the identifier of the process that most recently reached a breakpoint. Similarly, in protected mode PR is automatically reset to the appropriate process number at the occurrence of a fault or an exception.

When you invoke the Debugger from the Executive or from the Context Manager (using **CODE-GO**) just before the execution of an application program begins, PR is set to the identifier of the first process in the program.

All Debugger commands interpret the process with which they are associated as the current process. For example, whenever processor registers are read or written to, the registers of the current process are used.

Because PR is an internal register, you can set PR to the process identifier number [or the protected mode Task State Segment (TSS) selector] for any system or application process you need to debug without affecting the results of program execution. You will be introduced to the procedures for setting PR later in this section. First, however, the meaning of PR is discussed.

Meaning of PR

Based on the process identifier number (pid) to which PR is set, PR makes available the following information associated with that process:

- It implies a load offset to be used when loading symbols.
- It indicates which set of saved registers is to be displayed or modified by the Debugger.
- It indicates which process should have instructions executed individually (single stepped). (Single stepping is accomplished by the **CODE-X** command, which is described in Section 8, "Using Breakpoints.")
- It determines how addresses are to be interpreted. (For details on the differences between real mode and protected mode addressing schemes, see *CTOS / Open Programming Practices and Standards.*) Which type of address interpretation is currently in effect is reflected by the appropriate real mode or protected mode prompt. (For details on the different types of prompts, see " Debugger Prompts" in Section 4, "General Purpose Functions and Features.")

To view all the processes and their associated identifiers in the system, you can use the **CODE-S** command described in Section 7, "Display Commands."

In protected mode, PR has additional functionality. PR can also be set to a *Task State Segment* (TSS) selector. A TSS selector corresponds to the descriptor that contains the address of the TSS. There is a *TSS* for every protected mode *task* (that is, every process as well as every interrupt handler) in the system. The **CODE-S** command for protected mode (described in Section 7, "Display Commands.") allows you to view all these TSS's as well as all process identifiers.

The TSS provides you with additional information that you may find useful particularly if you are doing system debugging. As an example, the TSS's make available to you a complete set of saved registers for each interrupt handler in the system. In real mode, you are not provided this same convenience because there is no comparable structure that contains this information.

If you are debugging a real mode program executing on a protected mode operating system, you may encounter a situation in which it is useful to examine the TSS. As an example, if your program calls a protected mode system-common procedure such as PutFrameChars, it is possible that your program can generate a protected mode general protection (GP) fault.

If you set PR to the process identifier, your examination would be limited to real mode process execution up to the point at which the system-common procedure is called. The actual code for the system-common procedure, however, executes in protected mode, because it resides in the operating system itself. Your program, therefore, executes as a protected mode process for the duration of the routine. (This is accomplished by a technique called *aliasing*, which is described and illustrated in *CTOS / Open Programming Practices and Standards.*)

In this situation, you should approach debugging by first examining the parameters you passed to the system-common procedure. Examples of how to examine the parameters to a CTOS operation are provided in the tutorial in Section 3, "Debugging Session," as well as in Appendix D, "Debugging Tips." If, however, you are unsuccessful at detecting any problems with the parameters passed, you can set PR to the TSS. By doing so, you are able to examine the protected mode portion of the process.

Examining and Modifying PR

To debug a single-process program, you normally do not need to be concerned about the PR at all. However, if you intend to debug a multiprocess program, you must set PR to the process you want to debug.

As mentioned earlier, you can obtain the pids for all processes currently active in the system by using the **CODE-S** command described in Section 7, "Display Commands." From the CODE-S display, select the pid (or TSS) for the process you want to set PR to.

You set PR in the same way you would change any other word location: by opening the location and then entering a new parameter. If, for example, the current process is number 4, but you want to debug process number 7, you type

PR (press **RIGHT ARROW**)

The Debugger responds by displaying

```
PR → 4
```

Type 7 to the right of 4 as shown below:

```
PR → 4 7
```

Press **RETURN**. As a result, you change the current process number to process 7.

By setting PR to the appropriate pid or TSS, you then can use the **CODE-F** command to load the symbol file for that process. Information on setting PR and loading symbols for debugging crash dump files is included in Section 11, "The Executive Command: Debug File."

Processor Registers

Unlike the PR, which is located in memory, the processor registers are located in the central processing unit. The purpose of registers is to contain addresses and data obtained from memory.

By looking at register contents, you can verify whether they contain the addresses or data you would expect at a certain point in your program's execution. (For details on the processor register functions, see the Intel manuals listed in "Where to Go For More Information" in "About This Guide.") You can find practical examples of how some of the registers are used in Section 3, "Debugging Session," as well as in Appendix D, "Debugging Tips."

Registers are indicated in the Debugger by mnemonic symbols, such as

Mnemonic	Register
CS	Code Segment
DS	Data Segment
ES	Extra Segment
SS	Stack Segment

These registers, as well as the ones described below, are common to all processors:

- Registers with the mnemonics AX, BX, CX, and DX are all general registers.
- BP, BX, DI, and SI are general registers and index registers.
- FL contains flags.
- IP contains the instruction pointer.

For protected mode processors, there are additional registers. Table 6–1 lists all the processor registers and indicates which processors apply to each. Table 6–2 lists the internal Debugger registers.

You can read or write to any of the different types of processor registers. To do so, you indicate the mnemonic symbol of the register as a left-side value in a command you use to examine or to change the register contents. For example, you would type

AX (press **RIGHT ARROW**)

to display the value of a word in the AX register.

To see the contents of all registers, simply press **CODE-SHIFT-R**.

When debugging a multiprocess program, the processor register mnemonics indicate the machine registers associated with the current process.

Table 6–1. Processor Registers

Register Name	**Debugger Mnemonic** (if different)	**Processor** **8088 8086 80186**	**80286**	**80386**
AL		*	*	*
AH		*	*	*
AX		*	*	*
EAX				*
BL		*	*	*
BH		*	*	*
BX		*	*	*
EBX				*
CL		*	*	*
CH		*	*	*
CX		*	*	*
ECX				*
DL		*	*	*
DH		*	*	*
DX		*	*	*
EDX				*
SS		*	*	*
CS		*	*	*
DS		*	*	*
ES		*	*	*
FS				*
GS				*
SI		*	*	*
ESI				*
DI		*	*	*
EDI				*
BP		*	*	*
EBP				*
SP		*	*	*
ESP				*
IP		*	*	*
EIP				*

continued

Table 6–1. Processor Registers (cont.)

Register Name	Debugger Mnemonic (if different)	Processor 8088 8086 80186	80286	80386
FL		*	*	*
EFL				*
TR			*	*
MSW	MS			*
CR0	R0			*
CR1	R1			*
CR2	R2			*
CR3	R3			*
LD			*	*
SS0	S0		*	*
SS1	S1		*	*
SS2	S2		*	*
SP0	P0		*	*
SP1	P1		*	*
SP2	P2		*	*
LKB	LK		*	*

Table 6–2. Debugger Registers

Register Name	Debugger Mnemonic (if different)	Processor 8088 8086 80186	80286	80386
PR		*	*	*
CB		*	*	*
DB		*	*	*

Section 7
Display Commands

What are the Display Commands?

This section explains how to use Debugger commands to display

- (And print) the user screen and the Debugger screen
- Other contexts (under Context Manager/VM)
- A stack trace
- Linked-list data structures
- Process and exchange status
- Descriptor tables
- Request code definitions
- Flag bit mnemonics
- Request blocks
- Semaphore status

Displaying User Information: CODE-U

When the Debugger is running, it displays only the Debugger screen. The Debugger does not display the screen generated by the user process. To view the user screen without exiting from the Debugger, press **CODE-U**.

Once the user screen appears, press any key (except **CODE-U** again) to restore the Debugger screen.

You can display the status of a particular user by typing

userNum (press **CODE-U**)

where *userNum* is the user number. The display is an abbreviated form of **CODE-S**, with information about one user extracted. If the user number specified is for an application, the name of the run file displays also.

Below is an example of a display for a user whose user number is 4. The user number appears in the "user" column, which is second from the right. Notice that this example shows the name of the run file, "*Signon.Run.*"

```
Processes

id oPcb  cs      ip  st pr tss  ldt  ss    sp   ds   exch sgU  user partition
0E 7BB6 0560:01C7 80 14 0CB0 2CA0 0284:31E4 0284 104C 2648 1004 Primary
0F 7BC8 0560:01B0 81 7F 0CC0 2CA0 0284:3FB8 0284 104D 2648 1004 Primary

Status for Run File:  [Sys]<Sys>Signon.Run

Run Queue  |00|0F|55|

Exchanges

013 exchRqTracker        Messages
|2CB8:3FCA(1004,8800)|2648:744(1000,8A00)|2CB8:31FC(1004,8800)|
04C Primary              Processes  |0E|
04D Primary              Empty
```

For information about the contents of each column, see "Displaying Process and Exchange Status: CODE-S."

Printing the Debugger Screen: CODE-L

You can use the **CODE-L** command to print the dialogue that appears on the Debugger screen to a parallel printer. To activate this direct print feature, press **CODE-L.**

Note: CODE-L *requires that you have a parallel printer attached to your workstation. (For details on connecting the printer and setting it up, see the* CTOS Generic Print System Administration Guide.*) If the Generic Printing System (GPS) or the pre-GPS Spooler is installed, deinstall it before using* CODE-L. *Otherwise,* CODE-L *will not work.*

If the parallel printer is properly connected and is online, the following message appears on the screen:

```
Lpt echo ON
```

If the parallel printer is not properly connected, or if it is not online, the following message appears on the screen:

```
Lpt echo OFF
```

The **CODE-L** command toggles the echo of the Debugger output. If you press **CODE-L** again and again, the print feature alternates between its ON and OFF states.

You cannot use **CODE-L** if the parallel printer is also being used by the operating system or by the program that is being debugged.

With the Executive command **Debug File, CODE-L** takes the optional file name parameter

'filename' (press **CODE-L**)

where *'filename'* is the name of the byte stream file that the screen output duplicate is to be copied to.

Displaying Other Contexts

If you are using the Debugger under Context Manager/VM, you can switch to another context. This feature is useful for viewing your source code with an editor in one context while debugging your program in another. (For details, see "Invoking Context Manager/VM Operations" in Section 4, "General Purpose Functions and Features.")

Displaying a Stack Trace: CODE-T

You can use the **CODE-T** command to display the current sequence of procedure calls in your program. **CODE-T** causes the Debugger to trace the contents of your program's stack. For a complete stack trace to work properly, however, the stack must follow the rules for standard stack format. See Appendix C, "Stack Format," for details. If the stack is damaged, a stack trace usually contains invalid information. This can occur, for example, if a program erroneously overwrites a portion of the stack or uses the BP register for a value other than the stack pointer (SP). To circumvent this problem, you can use a variant of the **CODE-T** command to view only the valid portions of the stack. This variant is described later in this section.

Note: *BP must be used according to one of the two CTOS calling conventions. For details, see* CTOS/Open Programming Practices and Standards. *If you find that only a portion of the procedural call sequence is displayed (usually consisting of the current stack frame and one or two of the previous frames), it is possible that your compiler is not following BP conventions. See your language manual for details.*

Appendix C, "Stack Format," describes the most common usage of BP. Some languages must use mediators to conform to this format. For details on mediators, see the *CTOS Programming Guide.*

To display the procedure-invocation stack for the current process,

press **CODE-T**

The Debugger responds by displaying the entire stack. Note, however, that if any part of the stack is damaged, **CODE-T** displays only the valid portion beginning at the lowest stack address up to the point where the damage begins.

Starting the Trace at a Specified Address

As described earlier, **CODE-T** (with no parameters) displays the stack beginning at the lowest address (most current frame) and only displays the trace up to the point where BPs are no longer chained. To view a stack trace beginning at a different address,

type *addr* (press **CODE-T**)

where *addr* is the address of a valid BP. You can use this command, for example, to view valid portions of a damaged stack by specifying the next valid BP.

To display the stack for the *k* most recent active procedure invocations,

type *k, addr* (press **CODE-T**)

For example, to display the stack for the six most recent invocations, you type

6, SS:BP (press **CODE-T**)

A typical Debugger response is shown below:

```
0   3108     12D2:14A    (3,3,4F1F,3113)
1   3116     581F:95     (173C,1,4F1F,312A,4F1F,312E)
2   3130     5EC8:0C4    (4F1F,173C,4F1F,314D)
3   313E     571C:255    (26F0,47,4F1F,314D,4F1F,3158)
4   3160     5920:6E     (4F1F,2700,4F1F,3178,4F1F,31CD)
5   318E     5FA9:70     (2700,4E1F,0,600,4F1F,31CD)
```

The first line is the current frame. The second line is the previous frame, and so on. Each line displays the following information:

- The first column contains the level number for the stack frame described by that line.
- The second column contains the frame pointer (BP) for that frame.
- The third column contains the return address (CS:IP) that will be effective when control returns to that level.

The remainder of each line contains the parameters that were passed to the procedure corresponding to the stack frame. The current value of a parameter is shown if the parameter was passed by value. If, however, the parameter was passed by reference, the address of the value is shown.

Because the Debugger estimates the number of parameter words displayed (maximum 6), this number may not correspond to the number of parameters actually passed to the procedure. For more details on interpreting a stack trace, see Appendix C, "Stack Format."

Displaying the Trace in Short or Long Call Format

To display a stack trace in either short or long format,

type *assumeFar, k, addr* (press **CODE-T**)

where *assumeFar* is either 0 or 1. Zero implies a short call; one implies a long call. *k* is the number of most recent invocations. Here is an example of a request to display a stack trace in long call format:

1, 2, SS:BP (press **CODE-T**)

A typical Debugger response is shown below:

```
0   987E    0AC39:1C3    (842,2E98,9890,510,0B95,293)
1   987E    989A:2E98
```

Displaying Linked-List Data Structures: CODE-N

The **CODE-N** command displays linked-list data structures. To use **CODE-N**, the data structures must be linked with 16-bit pointers.

Note: *If the data structures are not linked with 16-bit pointers, use* CODE-D *to display the data structures manually.*

You should use **CODE-N** with the internal Debugger registers CB and DB. The following paragraphs explain how to do so.

To display a block of memory that is k bytes long that has a link word at the jth byte, first set CB equal to k, and then set DB equal to j, as shown:

CB → 0000 k

DB → 0000 j

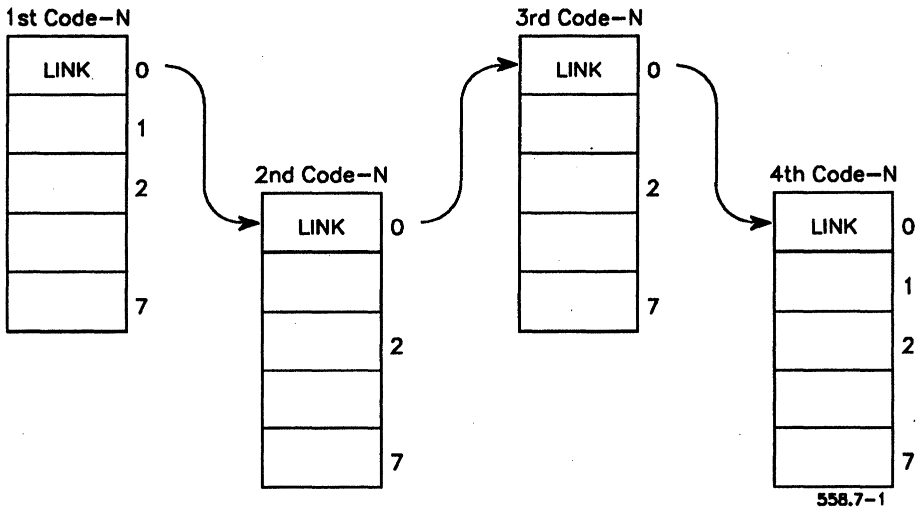
To display the first block,

type *addr* (press **CODE-N**)

To display each subsequent block, press **CODE-N** again.

Figure 7–1 illustrates this process.

Figure 7–1. Using CODE-N for Linked-List Data Structures



Or, to display *n* blocks at the same time,

type *n, addr* (press **CODE-N**)

where *n* is a decimal number.

For example,

3., 1217:3084 (press **CODE-N**)

specifies that three blocks are to be displayed at once. A typical Debugger response appears below:

```
1217:3084  02 31 C1 01 7C 3E AF 17
1217:3102  DA 31 83 80 68 26 76 5D
1217:31DA  00 00 C1 FF 04 4C 17 12
```

Displaying Process and Exchange Status: CODE-S

The **CODE-S** command displays the status of all processes and exchanges in the system. For protected mode, **CODE-S** also displays the status of interrupt handlers.

To obtain this information,

type **CODE-S**

The Debugger responds with the display of either two or three items. The first item is a list of all processes and pertinent data about them, and the second is a list of exchanges. The third item lists all interrupt handlers as well as all processes in the system.

To obtain only the third item,

type **0** (press **CODE-S**)

The **CODE-S** command is particularly useful for analyzing crash dumps. (Section 11, "The Executive Command: Debug File," provides examples of ways you can use **CODE-S** in crash dump analysis.)

Processes Display

The first item displayed by the **CODE-S** command is the Processes display. Figure 7–2 is an example of what this display looks like when using a real mode Debugger. An example of this display for protected mode is shown in Figure 7–3.

Each line in the displays shown in Figures 7–2 and 7–3 corresponds to a process that is currently active in the system. The number at the beginning of each line is the process identifier. To debug a process, you can set the Debugger internal process register (PR) to the identifier for that process. (For details on PR, see Section 6, "Working With Registers.")

Figure 7–2. Real Mode Operating System Processes Display

```
Processes

id oPcb cs        ip st pr ss        sp   ds   exch sgU  user partition
00 6BBA 0298:08B2 C0 02 04C0:99BE 04C0 0007 0000 0000
01 6BCC 0298:08B2 C0 01 04C0:9A70 04C0 000D 0000 0000
02 6BDE 0298:08B2 C0 04 04C0:9B6C 04C0 000E 0000 0000
03 6BF0 0298:08B2 C0 05 04C0:9D62 04C0 0010 0000 0000
04 6C02 0298:08B2 C0 06 04C0:9E5C 04C0 0012 0000 0000
05 6C14 0298:08B2 C0 08 04C0:9F5C 04C0 0013 0000 0000
06 6C26 0298:08B2 C0 07 04C0:A0F2 04C0 0014 0000 0000
07 6C38 0298:08B2 C0 04 0D78:03BE 0D78 001B 0000 0000
08 6C4A 0298:08B2 C0 03 0D78:7D22 0D78 0000 0000 0000
09 6C5C 0298:08B2 C0 06 12C0:FFFA 12C0 001C 0000 0002 Vdm_Ch 2.0
0A 6C6E 0298:08B2 80 14 01A4:96F2 01A4 001E 1190 0003 Primary
0B 6C80 0298:08B2 80 80 045C:FEB4 045C 0022 1598 0005 CM01
0C 6C92 0298:08B2 80 13 01A4:887A 01A4 0023 1190 0003 Primary
0D 6CA4 0298:08B2 80 0C 01A4:8A96 01A4 0025 1190 0003 Primary
0E 6CB6 0298:08B2 80 7F 045C:BBD5 045C 0027 1598 0005 CM01
0F 6CC8 F168:001D 80 78 F58B:0BAC F58B 002B 1650 0006 CM02
47 70B8 0318:0018 C1 FF 04C0:97DC 04C0 0000 0000 0000

Run Queue |47|
```

Figure 7–3. Processes Display for Protected Mode

```
Processes

id oPcb  cs   ip    st pr tss  ldt   ss   sp    ds   exch sgU  user partition
00 6BBA 0298:08B2 C0 02 0880 0000 04C0:99BE 04C0 0007 0000 0000
01 6BCC 0298:08B2 C0 01 0890 0000 04C0:9A70 04C0 000D 0000 0000
02 6BDE 0298:08B2 C0 04 08A0 0000 04C0:9B6C 04C0 000E 0000 0000
03 6BF0 0298:08B2 C0 05 08B0 0000 04C0:9D62 04C0 0010 0000 0000
04 6C02 0298:08B2 C0 06 08C0 0000 04C0:9E5C 04C0 0012 0000 0000
05 6C14 0298:08B2 C0 08 08D0 0000 04C0:9F5C 04C0 0013 0000 0000
06 6C26 0298:08B2 C0 07 08E0 0000 04C0:A0F2 04C0 0014 0000 0000
07 6C38 0298:08B2 C0 04 08F0 0000 0D78:03BE 0D78 001B 0000 0000
08 6C4A 0298:08B2 C0 03 0900 0000 0D78:7D22 0D78 0000 0000 0000
09 6C5C 0298:08B2 C0 06 0910 0000 12C0:FFFA 12C0 001C 0000 0002 Vdm_Ch 2.0
0A 6C6E 0298:08B2 80 14 0920 1498 01A4:96F2 01A4 001E 1190 0003 Primary
0B 6C80 0298:08B2 80 80 0930 15B0 045C:FEB4 045C 0022 1598 0005 CM01
0C 6C92 0298:08B2 80 13 0940 1498 01A4:887A 01A4 0023 1190 0003 Primary
0D 6CA4 0298:08B2 80 0C 0950 1498 01A4:8A96 01A4 0025 1190 0003 Primary
0E 6CB6 0298:08B2 80 7F 0960 15B0 045C:BBD5 045C 0027 1598 0005 CM01
0F 6CC8 F168:001D 80 78 0970 0001 F58B:0BAC F58B 002B 1650 0006 CM02
47 70B8 0318:0018 C1 FF 0CE0 0000 04C0:97DC 04C0 0000 0000 0000

Run Queue |47|
```

The Run Queue shown at the bottom of each display lists the processes that are in the ready state (that is, ready to run) in priority order. (For details on the states of a process, see your operating system manual.) The process scheduled next for execution is identified by the leftmost process identifier number in the queue.

The column headings in Figures 7–2 and 7–3 are defined below.

id	The process identifier number.
oPcb	The address of the Process Control Block for that process. The address is relative to the operating system's data segment.
cs ip	The address of the next instruction to be executed by the process.
st	A byte containing status flags. For details, see Figure 7–4.
pr	The priority of the process.

tss	(Protected mode only.) The Task State Segment (TSS) selector of the process.
ldt	(Protected mode only.) The Local Descriptor Table (LDT) of the process. A value of zero means the process is based in the Global Descriptor table. A value of 1 means the process is a real mode process executing on a protected mode operating system.
ss sp	The address of the top of the stack for the process.
bp	The base pointer for the process.
ds	The data segment of the process.
exch	The default response exchange for the process.
sgU	The selector of the User (U) Structure. A U structure contains information about a context (user number), such as the Extended Partition Descriptor and the User Control Block.
user	The user number for the process.
partition	The name of the partition in which the application or system service is loaded.

Figure 7–4 describes the meanings of the bits in the process status word.

Figure 7–4. Example of Process Status Word

Bit	Meaning When Set
0	Process is on the run queue
1, 2, 3, 4, 5	Process is suspended
6	Process is a system service
7	PCB for process is valid

Exchanges Display

The next item displayed by the **CODE-S** command is the Exchanges display. An example of an Exchanges display is shown in Figure 7–5.

Figure 7–5. Exchanges Display

```
Exchanges

003 SysIn               Processes   |09|
004 FilterPros          Processes   |07|
005 Termination         Processes   |06|
006 Sched               Processes   |05|
00A LclFileSys          Processes   |08|
00C Agent               Processes   |04|
010 exchPeriodic        Processes   |0A|
011 MstrAgentRcv        Processes   |03|
013 exchRqTracker       Messages
|2CB8:3FCA(1004,8800)|2648:744(1000,8A00)|2CB8:31FC(1004,8800)|
016 exchPit             Processes   |01|
019 SyncClockPro        Processes   |02|
024 RemoteCache         Processes   |0B|
02A Vdm_Ch    3.1       Processes   |54|
02C Vdm_Ch    3.1       Processes   |0C|
02E RKVS                Processes   |0D|
04C Primary             Processes   |0E|
```

Each line in the Exchanges display describes the state of an exchange. From left to right on each line, the following information is displayed:

- An exchange identifier
- The exchange name
- An indicator of whether processes or messages are queued at the exchange
- Either a list of the queued processes or a list of pointers to queued messages

For a display of processes or of the pointers to messages waiting on exchange *k*,

type *k* (press **CODE-S**)

For example, to obtain a display of the processes or of the pointers to messages waiting on exchange 7, you would type

7 (press **CODE-S**)

Typical Debugger responses are shown below. The first is for processes, and the second is for messages.

```
07 - Processes |03|

07 - Messages |ADB2:1217|
```

Results are the same whether you use **CODE-S** or **CODE-SHIFT-S**.

Tasks Display

For protected mode, a third item is displayed by the **CODE-S** command. This item shows all interrupt handlers and processes (collectively called *tasks*) in the system. An example (portion) of this display is shown in Figure 7–6.

Figure 7–6. Tasks Display

```
sgTss/pid      link   cs:ip

0960                0430:0059 FinishCrash
0978                0430:0395 DoubleTaskFault
0990                0430:03A6 StackTaskFault
09A8                0430:03AF PageTaskFault
09C0      busy 0BD0 0430:0495 GPTaskFault
09D8                0630:0060 TraceRawInt
09F0                0430:03BF NmiTaskFault
.
.
.
0B28                0528:0000 LpInterrupt
0B40                0410:0006 RawCommNubCD
0B58                0410:000B RawCommNubEF
0B88                0448:033E SoftwareDmaInterrupt
.
.
.
```

The column headings in Figure 7–6 are defined below. From left to right, these are

sgTss	The Task State Segment (TSS) of the process. A TSS contains the complete register context of a process in protected mode. For details, see the Intel manuals listed in "Where to Go For More Information" in "About This Guide."
pid	The process identifier number of a process.
link	The contents of the TSS "back link" field. For details, see the *80386 Programmer's Reference Manual.*
(busy)	This column indicates whether or not the process is busy. For details, see the *80386 Programmer's Reference Manual.*
cs:ip	The address of the next instruction to be executed by the process.
(symbol)	The symbolic address (optional) of a system task. Before using **CODE-S** to display this column, you must set the Debugger internal process register (PR) to the process identifier number of a system process (such as 1), then load the operating system symbol file, *[Sys]<Sys>SysImage.sym.*

Displaying Descriptor Tables: CODE-V

For protected mode, you can use the **CODE-V** command to display the contents of the Local Descriptor Table (LDT), Global Descriptor Table (GDT), or Interrupt Descriptor Table (IDT). You can also use the **CODE-V** command for a very different purpose when you use it with the single parameter fl. This special use is described in "Displaying Flag Mnemonics: FL CODE-V."

Using **CODE-V** to display descriptor tables is particularly useful when you need to find out why your protected mode program generated a general protection (GP) fault. If your program faulted, you can use **CODE-V** to examine the selector in question, to see if the selector is valid and examine the descriptor limit, to determine whether the segment limit has been exceeded. Another less common use of **CODE-V** is to look at the IDT to determine which Task State Segment (TSS) handles an interrupt. You would only use **CODE-V** to look at the IDT if you are debugging interrupt handlers.

You can use the **CODE-V** command with no parameters or with any of several optional parameters. To display a descriptor table,

type *table, no. Sns, startSn* (press **CODE-V**)

where

table	Is one of the following values: 0=GDT 1=LDT 2=IDT
no. Sns	Is the number of selectors
startSn	Is the starting selector

If only one parameter is given (and it is not the special value fl, which is described later in this section), the parameter is interpreted as the starting and only selector (Sn). In this case, the entry is displayed, whether or not it is valid, and the user number that owns the selector is also displayed.

If two parameters are given, they are interpreted respectively as the number of selectors to examine and the starting selector. If three parameters are given, they are interpreted as the type of descriptor table, number of selectors, and starting selector, in that order. In both these cases, **CODE-V** shows only the valid descriptor table entries. Therefore, the number of entries shown may be smaller than the number of entries that you specified.

For example, to display descriptors of the GDT, you could type

0, 30, 4a0 (press **CODE-V**)

The Debugger would return a display such as the one shown in Figure 7–7.

Figure 7–7. GDT Display

```
GDT

iSN  Sn   base     limit ar p
0094 04A0 011CF280 00100 9B 0 286 code, non-conforming, readable
0095 04A8 0101DB50 0001F 92 0 286 data, expand up, writable, not accessed
0096 04B0 0101DB70 0001F 92 0 286 data, expand up, writable, not accessed
0097 04B8 0100SAB0 0D7CB 93 0 286 data, expand up, writable
00C2 0610 17A0:80360006 00 84 286 call gate
00C3 0618 0298:000E0E98 00 84 286 call gate
```

The column headings in Figure 7–7 are defined below. From left to right these are

iSn	The nth selector (Sn divided by 8)
Sn	The selector
base	The base address
limit	The limit of the descriptor
ar	The access rights
p	The protection level; range is 0 through 3

The remaining material to the far right is an English version of the access rights.

If the selector you are examining is a gate, the display differs as follows:

```
iSN   Sn    Sn:ra      wc   ar   p
0072  0394  02E8:0C48  02   E4   3   call gate
```

where all definitions are as above, except wc is the count of words copied from the caller's stack to the called procedures stack.

Displaying Flag Mnemonics: fl CODE-V

You can also use the **CODE-V** command with the single parameter fl to display the flags register mnemonics. To do so,

type **fl** (press **CODE-V**)

The Debugger displays the mnemonics for each flag bit set in the flags register. A typical Debugger response is

```
if tf pf iopl=0
```

For details on the flags register, see the *iAPX 286 Programmer's Reference Manual,* the *80386 Programmer's Reference Manual,* and the *i486 Microprocessor Programmer's Reference Manual.*

Displaying Request Definitions: CODE-W

To display information about how a request is defined, you can use the **CODE-W** command. This command is typically used by the systems programmer who is debugging a request-based system service program. For details on requests and request-based system services, see the *CTOS Operating System Concepts Manual* and the *CTOS Programming Guide.*

Using **CODE-W** with no parameter (the default), you can display information for all request codes.

Press **CODE-W**

You can display information about a single request:

type *r* (press **CODE-W**)

where *r* is the request code number. For example, to display request information defining request code 0A, you type

0A (press **CODE-W**)

The request code number does not have to be in hexadecimal. The following request code is the equivalent of 0A and will produce the same result:

10. (press **CODE-W**)

A typical workstation Debugger response is shown below:

```
RC      Exch    LSC     NetRouting
000A    000A    000B    0E
```

From left to right, the meanings of the fields displayed are as follows:

RC	The request code
Exch	The identifier of the system service exchange for the system service serving this request code
LSC	The local service code
NetRouting	The network (for example, B-Net II or CT-Net™) routing code for this request

For shared resource processors, the display would look like this:

```
Rq Code          Exch     LSC      ICC    Net
0023(    35.)    0808     0000     23     88       Read
0024(    36.)    0808     0001     23     88       Write
```

where the fields are the same as in the previous display, with the following new information:

Rq Code	The request code and its decimal equivalent in parentheses.
ICC	The Inter-CPU-communication routing information for the XE-530. For details on ICC, see the *CTOS Operating System Concepts Manual.*
Net	The routing request information. For details, see the *CTOS Operating System Concepts Manual.*

You can also use **CODE-W** with two parameters to display several lines, each of which contains information about a different request. To display several lines starting with request information for request code *r*,

type *n, r* (press **CODE-W**)

where *n* is the number of lines to be displayed. For example, to display six lines starting with request code 0A, you type

6, 0A (press **CODE-W**)

Displaying Request Blocks: CODE-Q

The **CODE-Q** command displays the contents of a request block.

Type *addr* (press **CODE-Q**)

where *addr* is the SA:RA of the request block.

Displaying Semaphore Status: CODE-Y

The **CODE-Y** command displays the status of currently active semaphores.

Type **CODE-Y**

Below is an example of the output of **CODE-Y**.

```
  sem              owner              threads
handle    state   user/pr    type   waiting
23600003  locked  0000/000D   03                  "Semaphore Lock"
2360006C  clear   0002/005E   03
2360009C  clear   000D/001F   04
2360012C  clear   000F/0026   00
236009fc  locked  000E/001F   00      0024      "SystemDLL"
```

The semaphore status information is described below.

sem handle	The semaphore handle.
state	Shows if the handle is locked or clear.
owner user/pr	Shows the user number and process id of the owner.
type	Shows the value describing the semaphore. For more information about semaphore types, see the SemOpen request in the *CTOS Procedural Interface Reference Manual.*
threads waiting	Shows the threads that are waiting and, optionally, their names (in the last column).

Displaying DLL Status: CODE-Z

The **CODE-Z** command displays the currently active dynamic link libraries and their handles.

Type **CODE-Z**

Here is an example of the output of the **CODE-Z** command.

```
id        name              files

4980      SysMono.font      [klm]<pmdll>SysMono.fon
4950      Time.font         [klm]<pmdll>Times.fon
4900      Helv.font         [klm]<pmdll>helv.fon
4700      PmMle             [klm]<pmdll>PmMle.dll
3F98      HelpMgr           [klm]<pmdll>HelpMgr.dll
3AA8      PmGre             [klm]<pmdll>PmGre.dll
3BB8      PmWin             [klm]<pmdll>PmWin.dll
3CD0      Display           [klm]<pmdll>lowRes>display.dll
3948      System            [klm]<pmdll>System.dll
3E28      SesMgr            [klm]<pmdll>SesMgr.dll
```

The DLL status information is described below.

id	The DLL identification.
name	The name of the DLL.
files	The name of the DLL handle.

Section 8
Using Breakpoints

What is a Breakpoint?

A breakpoint is a user-defined location in code. When a process reaches a breakpoint, the process is suspended, and the Debugger is entered.

If the Debugger is operating in simple mode, all user processes are suspended whenever any breakpoint is taken. If the Debugger is in multiprocess mode, however, only the process that has taken the breakpoint is suspended. (See Section 9, "Debugger Modes," for a description of the Debugger operating modes.)

Setting a Breakpoint at an Instruction

Although the examples in this section show you how to set a breakpoint at an address, you can also set a breakpoint at the instruction located at the address. In fact, setting a breakpoint at an instruction is more commonly done.

As an example, to set a breakpoint at the instruction located at TYPESUB+2B, you first display the instruction by pressing **MARK**. **MARK** appears as the right triangle in the Debugger display as shown below:

```
TYPESUB+2B◣
```

As a result of pressing **MARK**, the Debugger displays the instruction at that location, for example

```
CALL OPENBYTESTREAM
```

Then, you enter a period (.) followed by one of the breakpoint commands described in this section. If, for example, you set a breakpoint using the **CODE-B** command (described next), the Debugger display would appear as

```
TYPESUB+2B MARK CALL OPENBYTESTREAM    .^b
```

The period (.) means "at the address of the instruction, CALL OPENBYTESTREAM."

You will often set breakpoints in this manner because you typically view instructions using the **DOWN ARROW** until you find the one where you want to set the break. For examples of how this is done in the context of a debugging session, see the tutorial in Section 3, "Debugging Session."

Setting and Querying Breakpoints: CODE-B

You can use the **CODE-B** command to set an *unconditional* breakpoint. An unconditional breakpoint does not depend on the evaluation of a relational condition for the process to be suspended at the break. Later in this section, you will be introduced to conditional breakpoints.

To set an unconditional breakpoint, press **CODE-B** preceded by one parameter. For example, to set a breakpoint at the address *addr*, you can type

addr (press **CODE-B**)

(It is more common, however, to press **MARK** to designate a location, then type a period (.) and press **CODE-B**.)

You can use **CODE-B** to set a breakpoint in an overlay, even if the overlay is not present in memory. (See Section 10, "Overlays," for details on debugging overlays.)

A breakpoint stays in effect until you remove it explicitly (by using the **CODE-C** command) or until the process terminates.

When a program in a memory partition calls the Chain or Exit operation, or is otherwise terminated, only the breakpoints in that partition are removed.

You can also use the **CODE-B** command to query breakpoints you have already set during a debugging session. To display a list of all of the breakpoints that are set at any given time,

press **CODE-B**

Clearing Breakpoints: CODE-C

As you are debugging your program, you will want to clear unnecessary breakpoints. It is a good idea to clear all breakpoints, for example, before you exit the Debugger.

To clear a breakpoint at a given address, type the address of the breakpoint to be cleared, then press **CODE-C**. For example, to clear the breakpoint at address *addr*,

type *addr* (press **CODE-C**)

To clear all breakpoints,

press **CODE-C**

Proceeding From a Breakpoint: CODE-P

When you have checked the code at a breakpoint, you can move on to the next breakpoint. To proceed from the most recently found breakpoint in the current process without clearing the most recent breakpoint,

press **CODE-P**

The breakpoint remains in effect, and the process continues. If the process was not broken by the breakpoint, the Debugger ignores the **CODE-P** command. (In this case, because the process is still running, you cannot logically command it to resume running.)

To remove the breakpoint before proceeding,

type **0** (press **CODE-P**)

To proceed, and to break the *k*th time the breakpoint is reached (instead of the next time it is reached),

type *k* (press **CODE-P**)

where *k* is a decimal number. **CODE-P** with no parameters is equivalent to **CODE-P** with a parameter of 1.

If you entered the Debugger by pressing **ACTION-A**, you can press **CODE-P** or **GO** to exit. If, however, you entered by pressing **ACTION-B**, you must press **GO** to exit. (For details, see Section 9, "Debugger Modes.")

Setting Conditional Breakpoints: CODE-A

You can use the **CODE-A** command to set a conditional breakpoint. A *conditional* breakpoint is a breakpoint that is associated with a relational condition. When a process reaches the breakpoint, the process is suspended only if the relational condition evaluates to TRUE (0FFh).

To set a conditional breakpoint, type an address, then press **CODE-A**. For example,

type *addr* (press **CODE-A**)

Using the Patch Area to Define a Relational Condition

You define the relational condition in the Debugger's patch area. The *patch* area is a 50-byte space reserved for defining conditional breakpoints and addressed by the symbol, PatchArea. To use the patch area,

type **PatchArea** (press **MARK**)

The Debugger displays the first instruction in the patch area. For example, a typical Debugger response is

PatchArea **MARK**   NOP

At the end of this line, the Debugger displays the space prompt. At the prompt, you can begin defining the relational condition by typing your first instruction. For example, in response to the line displayed above by the Debugger, you could type

MOV AX, WORD PTR [0]

To display and modify the next instruction in the patch area, press **DOWN ARROW**. The Debugger displays

```
PatchArea+3 MARK    NOP
```

Then type your instruction at the prompt.

Repeat the procedure of pressing **DOWN ARROW** and typing an instruction until you have completely defined the relational condition in the patch area.

Say, for example, you want the Debugger to take the breakpoint if the value of memory location DS:0 is 200h. After entering all instructions, the patch area display would appear as follows:

```
PatchArea MARK    NOP  MOV  AX,  WORD  PTR  [0]
PatchArea+3 MARK    NOP  CMP  AX,  200
PatchArea+6 MARK    NOP  JE  .+5
PatchArea+8 MARK    NOP  MOV  AL,  0
PatchArea+0A MARK    NOP  DEBUG
PatchArea+0B MARK    NOP  MOV  AL,  OFF
PatchArea+0D MARK    NOP  DEBUG
```

Instructions that you add to the patch area must set the register AL to TRUE (0FFh) if the breakpoint is to be taken. Otherwise, your instructions must set AL to FALSE (0h).

The preceding patch area example illustrates the setting of AL. In the display, the first instruction places the value at DS:0 into AX. Then, the second instruction compares the value in AX with 200. If the value equals 200, the third instruction jumps (5 bytes) to the sixth instruction, which sets the register AL to 0FFh. If, however, the value in AL does not equal 0, the fourth instruction sets AL to 0h.

So that control will return to the Debugger, the last instruction in the relational condition must be an INT 3 instruction.

After the condition is evaluated, only the original value of the AX register is restored. Any of the registers should be saved (pushed) before use and restored (popped) before the INT 3 instruction is executed.

Setting Multiple Conditional Breakpoints

You can set more than one conditional breakpoint by specifying an additional parameter for the **CODE-A** command. This parameter specifies the PatchArea offset at which the relational condition begins.

For example, to set a conditional breakpoint at the symbolic address Initialize, whose relational condition begins at PatchArea+20, you type

20, 0, INITIALIZE (press **CODE-A**)

The optional 0 parameter indicates that it is an instruction breakpoint. If 20 is specified, 0 must also be specified. (The reason for specifying 0 is explained in "Setting Data Breakpoints: CODE-B, CODE-A Variant.")

Changing Unconditional to Conditional Breakpoints

You can change an unconditional (**CODE-B**) breakpoint into a conditional breakpoint at any time, by typing

addr (press **CODE-A**)

where *addr* is the address of the unconditional breakpoint.

After entering this command, you must also add the conditional code in the PatchArea.

Setting Data Breakpoints: CODE-B, CODE-A Variant

A variant of the **CODE-B** command is used to set data breakpoints on 80386-based operating systems. This variation is used to execute a breakpoint when the data at a specific location changes because an instruction was executed that read or wrote to this location.

To set a data breakpoint in a program running on an 80386-based system,

type *k, n, addr* (press **CODE-B**)

where *k* is 0 or 1. A value of 0 indicates break on a read or write to the location; 1 means break only on a write to the location. This parameter is optional, with 1 as the default. *n* specifies the length of the data item being changed, and *addr* is the address. Acceptable values of *n* are

- 1 = byte
- 2 = word
- 4 = double word

For example, the command

2, DS:0 (press **CODE-B**)

instructs the Debugger to break when the word at location DS:0 is written.

Note that conditional data breakpoints can be set using the same syntax and the **CODE-A** command.

Starting a Process at a Specified Address: CODE-G

The foregoing commands always cause a process to start executing from the last breakpoint address. To begin process execution at a different address,

type *addr* (press **CODE-G**)

where *addr* is a segmented address. The address can be either a user-defined public symbol or a logical address of the form

x:y

where *x* is an appropriate CS parameter, and *y* is an appropriate IP parameter.

Executing Instructions Individually: CODE-X

You can use the **CODE-X** command to execute instructions individually. This technique is called *single stepping*. You typically single step through small sections of a program. You can, for example, set an unconditional breakpoint using **CODE-B** to execute code up to the location where you want to start single stepping using **CODE-X**.

To execute the next instruction in the current process,

press **CODE-X**

After this instruction is executed, the next instruction is opened and displayed. Thus, you can press **CODE-X** repeatedly to see a series of instructions displayed and executed one by one.

To resume continuous execution of instructions after using **CODE-X**, you can press either

CODE-P or **GO**

Whenever you use the **CODE-X** command to execute an instruction that loads a segment register, two instructions are actually executed.

For example, in the portion of code that follows, if you use **CODE-X** to execute the instruction LES BX, [bp + 6], the instruction PUSH ES is also executed, and the PUSH BX instruction is displayed at the open location, as indicated:

```
LES BX, [bp + 6]
PUSH ES
PUSH BX
```

If you are using the real mode Debugger to debug an application under the Context Manager, you can enhance performance when using **CODE-X** by editing the Context Manager configuration file. For details, see Appendix F, "Configuration Options for the Debugger."

Note: *With interrupts disabled, you cannot use* CODE-X *to single step instructions because the interrupt for single stepping is ignored.*

Under certain circumstances, executing instructions individually is not recommended. For example, you should not single step through operating system routines. Instead, set a breakpoint following the return from the routine. One convenient way to do this is to use the **CODE-E** command, which is described below.

Breaking After the Current Instruction: CODE-E

CODE-E lets you step over an instruction and break after it. To break after the current instruction,

press **CODE-E**

Executing the **CODE-E** command is equivalent to executing the following command series:

Press **DOWN ARROW** (↓)

Type a period (.)

Press **CODE-B** (**^b**)

Press **GO**

The following portion of a Debugger script demonstrates how to use **CODE-E**:

```
TEST1+2A MARK  CALL OPENBYTESTREAM    .^e
TEST1+2F MARK  PUSH AX
Exiting Debugger
Break at TEST1+2F in process 0D
```

By using **CODE-E** as shown, the Debugger executes the entire OpenByteStream procedure and breaks at the instruction, PUSH AX, following the return. At this point, you could conveniently check AX to see the error code returned by OpenByteStream.

In contrast, the **CODE-X** command steps through, not over, an instruction.

With certain Debugger versions (noted in Appendix H), **CODE-E** clears the breakpoint after breaking at the instruction.

Note: *You cannot use* CODE-E *to break after an instruction that loads a segment register.*

Setting Breakpoints in Interrupt Handlers: CODE-I

The breakpoint commands described thus far in this section let you place breakpoints in normal user code and in normal operating system code. However, these commands alone cannot place a breakpoint in the operating system Kernel or in an interrupt handler. Since there may be times when you want to set a breakpoint in an interrupt handler or in the same code where interrupts are disabled, the Debugger provides the **CODE-I** command.

To set a breakpoint at address *addr* in an interrupt handler or in the operating system Kernel,

type *addr* (press **CODE-I**)

The standard keyboard and video facilities of the operating system support your interaction with the Debugger, except when the operating system Kernel or an interrupt handler is broken. At these breakpoints, all processes (including operating system processes) are suspended, and the Debugger then works by direct access to the physical keyboard and the screen.

When using **CODE-I** with the real mode Debugger under the Context Manager, you need to reserve enough memory for the Debugger to always be memory resident. Otherwise, if Debugger swapping is allowed, you may get an error message indicating that there is not enough memory to swap in the Debugger. (For details on how to configure the Context Manager, see Appendix F, "Configuration Options for the Debugger.")

CODE-I breakpoints are prohibited if the Debugger has swapped out a part of the user program. The Debugger does this automatically when the user program and the Debugger together are too large to fit in the partition. (For details, See Appendix E, "Debugger Swapping.")

Before exiting from interrupt mode, you should explicitly remove any **CODE-I** breakpoints by using the **CODE-C** command.

The Debugger uses hardware interrupts 1 and 3. Therefore, user programs should not service hardware interrupts 1 or 3. Other parts of the operating system use other hardware interrupts. (For a description of all the interrupt types, see *CTOS / Open Programming Practices and Standards*.)

Assembly Language Calls: The INT 3 Instruction

The Debugger is automatically entered when an INT 3 instruction is executed.

If, for example, you coded the INT 3 instruction at location 9904:6A in your program, upon execution of that instruction, the system would enter the Debugger and display the message

```
Debugger call at 9904:6B in Process 8
```

Note that the address displayed in this message (9904:6B) is located 1 byte after the INT 3 instruction (9904:6A).

In the Debugger, if you typed the address of the INT 3 instruction and pressed **MARK**, the Debugger would disassemble the instruction by displaying the DEBUG mnemonic, as shown below:

9904:6A **MARK** DEBUG

Execution of INT 3 causes an interrupt to occur. The effect of using this instruction is identical to setting a breakpoint. INT 3 is particularly useful for suspending a program at a predetermined point in its execution for debugging purposes. The instruction requires only 1 byte and can be substituted for any instruction opcode. The saved value of CS:IP points to the next instruction to be executed. (For details on how you can use the Debug File command to set an INT 3 instruction in your program, see "Patching a Run File" in Section 11, "The Executive Command: Debug File.")

Section 9
Debugger Modes

What are the Debugger Modes?

The Debugger operates in three modes: simple mode, multiprocess mode, and interrupt mode. Throughout this guide, these modes are referred to as *Debugger modes* to distinguish them from the processor modes (that is, real mode and protected mode) in which the Debugger also operates.

Simple mode is used to debug most programs. All user processes are suspended in this mode.

Multiprocess mode is used to debug a program whose operation depends on the continuous execution of all processes except those explicitly stopped at breakpoints.

Interrupt mode is used to debug interrupt handlers or for debugging that requires breakpoints be set when interrupts are disabled.

This section describes how these three modes are related to the different ways of entering the Debugger. In protected mode, the Debugger prompt for each of the Debugger modes is preceded by the appropriate protected mode symbol. (See "Debugger Prompts" in Section 4, "General Purpose Functions and Features," for details.)

Simple Mode

You can invoke simple mode by pressing **ACTION-A**. Simple mode also is invoked automatically when a **CODE-B** breakpoint is executed after **ACTION-A** invokes the Debugger.

You also can enter the Debugger in simple mode by pressing **CODE-GO** or using the Chain or LoadTask operations. (For details on these operations, see your operating system manual.)

In simple mode, all user processes are suspended when the Debugger is entered. This mode does not affect operating system services or interrupt handlers.

The nonresident portion of the real mode Debugger can swap in and out of memory. (See Appendix E, "Debugger Swapping.") Swapping, however, is transparent to the user. For details on the Debugger memory requirements, see your operating system release documentation.

In simple mode, the Debugger prompt is an asterisk (*).

Multiprocess Mode

You can invoke multiprocess mode by pressing **ACTION-B**. Multiprocess mode is invoked automatically when a **CODE-B** breakpoint is executed after **ACTION-B** invokes the Debugger.

In multiprocess mode, all user processes continue execution after you enter the Debugger. The Debugger suspends only those user processes that have reached a breakpoint.

Like simple mode, multiprocess mode does not affect operating system services or interrupt handlers. Multiprocess mode is most useful for debugging certain realtime programs, such as the timer process in the Executive.

If you invoke the Debugger by pressing **ACTION-B**, you can still press **ACTION-A** to invoke the Debugger in simple mode and suspend all user processes. However, once you invoke the Debugger in simple mode, you cannot change directly to the multiprocess mode.

The Debugger alone requires about 110K bytes of memory. (Check your operating system release documentation for the exact memory requirement.) The entire Debugger (including its nonresident portion), together with the program being debugged, must fit into memory. If memory is insufficient, a status message is displayed, and the Debugger switches to simple mode.

Provided enough memory is available, you can set a CODE-I breakpoint in multiprocess mode at any time. The Debugger goes into interrupt mode when a CODE-I breakpoint is taken. (See "Setting Breakpoints in Interrupt Handlers" in Section 8, "Using Breakpoints," for details on CODE-I breakpoints.)

If the current process as defined by the process register (PR) is not suspended, the Debugger prompt is the pound sign (#), indicating that the Debugger is in multiprocess mode.

If, however, the current process has been suspended, the Debugger prompt is an asterisk (*), indicating that the Debugger is in simple mode.

Proceeding and Exiting: CODE-P, CODE-G, and GO

In multiprocess mode, if the current process has been suspended, the **CODE-P** command causes that process to resume. The **CODE-P** command, however, does not exit from the Debugger. To exit, you must press **GO**.

Similarly, you press **CODE-G** to begin process execution at a different address from the one at which the process stopped. (Pressing **CODE-G** does not exit from the Debugger.) To use this feature,

> type *addr* (press **CODE-G**)

where *addr* is the logical or symbolic address at which you want to begin process execution following a breakpoint.

As with **CODE-P**, you press **GO** to exit from the Debugger.

Keyboard and Video Control

Pressing **GO** does not have the same effect in multiprocess mode that it has in simple mode. In simple mode, pressing **GO** swaps control of the keyboard and screen between processes; whereas, in multiprocess mode, pressing **GO** exits the Debugger and returns screen and the keyboard control to the user process.

Interrupt Mode

Interrupt mode occurs when a user process reaches a CODE-I breakpoint. In interrupt mode, the Debugger takes control and runs with interrupts disabled.

As in multiprocess mode, the entire Debugger must fit into the available memory. If you set a CODE-I breakpoint and enough memory is not available, a status/error message appears.

In interrupt mode, the Debugger prompt is an exclamation mark (!).

Section 10
Overlays

The Virtual Code Management facility permits you to configure a program into overlays. Only code can be overlaid, not data segments. This section describes how to debug an overlay.

Examining Code in an Overlay

An *overlay* is a part of a program that remains on a disk until it is called. The Debugger can display instructions that are contained in an overlay, whether or not the overlay is present in memory. (For details on overlays and the Virtual Code Management facility, see your operating system manual.)

To display instructions contained in an overlay,

type *symbolic addr* (press **MARK**)

The Debugger displays the symbolic address you typed plus the instruction at that address, for example

```
init+43 MARK      ADD  BYTE  PTR  [BP] [DI-46],CL
```

To continue displaying instructions beyond the first one, press **DOWN ARROW**. Each successive instruction is displayed, for example

```
{INIT+44} MARK      DEC  BX   DOWN ARROW
{INIT+45} MARK      MOV  DX, 9A50   DOWN ARROW
```

Note the braces enclosing the symbolic locations. Whenever you use the **DOWN ARROW** command to display the next instruction, the Debugger encloses the symbolic address of that instruction with braces if the instruction is contained in a nonresident overlay.

You cannot modify instructions contained in a nonresident overlay (although you can set breakpoints in such overlays). Patches that you make to an overlay in memory remain in effect only while that overlay is resident in memory.

Section 11
The Executive Command: Debug File

Uses of the Debug File Command

The Executive command Debug File lets you examine and modify the data in files.

This command is particularly useful for analyzing a crash dump file. A crash dump file contains a copy of the system memory image as it appeared just before a crash.

You can, however, use this command to examine and/or patch any run file. Although you would eventually need to correct your source file, at least you are saved from having to make the same corrections each time you run your program. When you patch a run file in the Debugger, on the other hand, you are only correcting a memory copy of the file. Therefore, it is necessary to make the same corrections each time your program is run.

As an example of patching a run file, you can use the Debug File command to set an INT 3 instruction at the beginning of your program to suspend execution at that point. Details on how to do this are described in "Patching a Run File."

Another use for the Debug File command is patching data files.

Invoking the Debug File Command

To invoke the Debug File command from the Executive, enter the command name on the Executive command line, and press **RETURN**. The following command form then appears:

```
Debug File
 File Name       ________________________________
 [Write?]        ________________________________
 [Image Mode?]   ________________________________
 [Symbol File]   ________________________________
 [Script File]   ________________________________
 [Output File]   ________________________________
```

where

File Name

Is the name of the file or device that you want to examine or modify.

[Write?]

Asks you if you want to modify any data in the file (default is no). If you enter yes, you can modify the data.

[Image mode?]

Asks you if you want the Debug File command to interpret the data in the run file (default is no). If you enter **yes**, the Debug File command interprets the data exactly as it appears in the run file. Otherwise, the command interprets the data the way it appears when loaded into memory.

[Symbol File]

Is the name of the symbol file. If examining a crash dump, this value can only be the name of the operating system symbol file. If examining a run file, this value can only be the name of the symbol file produced by the Linker.

[Script File]

Is the name of the script file. If you enter a name, the Debugger reads commands from this file rather than the keyboard.

[Output File]

Is the name of the output file. If you enter a name, the Debugger writes output to this file rather than the video display.

When invoked, the Debug File command prompts you with a percent sign (%).

Debugger Commands You Can Use

To examine and modify the data in a file, you can use all the Debugger commands except those involving run time execution. As an example, you cannot use **CODE-B** (to set and execute to breakpoints) or **CODE-P** (to proceed from breakpoints).

You can use **CODE-S** (to display process and exchange status) and **CODE-T** (to display a stack trace) only if the file being debugged is a crash dump file.

Exiting

To exit from the Debug File command, press either **FINISH** or **GO**. Any modifications you made to the data in the file are properly recorded on the disk only if you press **FINISH** to exit. If you want to exit without saving your changes, press **ACTION-FINISH**.

When you modify a run file, the Debug File command automatically corrects the run file checksum word.

Patching a Run File

You can use the Debug File command to patch a run file such that the Debugger is automatically entered at the program entry point. This patch is useful, for example, in a situation where you have a series of chained run files, and you need to suspend execution at the entry point of one of the chained files for debugging purposes.

Say, for example, a program consists of an installation file, *InstallProgram.run*, which sets up the program environment and chains to *Program.run*, the run file you need to debug. To execute code up to the entry point of *Program.run* and then enter the Debugger, you can set an INT 3 (Debug) instruction in *Program.run* by performing the following steps:

1. In the Executive, invoke the Debug File command, and complete the form as follows:

   ```
   Debug File
     File Name         Program.run
     [Write?]          y
     [Image Mode?]     
     [Symbol File]     
     [Script File]     
     [Output File]     
   ```

2. Press **GO** to execute Debug File.
3. At the prompt, type **CS:IP** (press **MARK**).

   The program entry point is displayed by the instruction

      MOV SP,BP

4. Press **DOWN ARROW** to view the next instruction,

   Debug File displays

      STI

   This is the Start Interrupts instruction.

5. Type **Debug** to the right of this instruction.
6. Press **RETURN**

   As a result, the STI instruction is replaced with the Debug instruction.

7. Press **FINISH** to record the patch you made to the file.

When the program is run, it executes up to the beginning of *Program.run*. Then, the Debugger is entered.

Examining a Crash Dump

As mentioned earlier in this section, you can also use the Debug File command to analyze a system crash dump. A *crash dump* is a byte-for-byte snapshot of what system memory looks like if you were able to enter the Debugger at the instant a system crash occurs. However, to examine a crash dump using the Debug File command, the system memory must be copied to a disk file.

Details on crash dump files are contained in the *CTOS System Administration Guide* and the *CTOS Executive Reference Manual.* Additional details for protected mode operating systems are contained in Appendix G, "Extended Crash Dump Process," in this guide. Details on the extended crash dump process, crash dump file size requirements, as well as ways you can conserve disk space are contained in Appendix G.

Usually there is an immediate reason for a system to crash. A process may have called the Crash operation or a (protected mode) general protection (GP) fault may have occurred, for example. In other cases, however, the source of the crash may have occurred at some point prior to the actual crash. In these situations, it may be necessary to try to reconstruct the conditions that led to the crash in the first place.

Certain program errors (for example, an invalid pointer) that cause a system crash under real mode CTOS will cause a protection fault under protected mode CTOS/VM 2.0 and later versions. Unlike system crashes, protection faults are not always fatal. If the faulting program is serving a request or is the operating system itself, the fault is fatal and the operating system executes a system crash.

The remainder of this section introduces techniques you can use to approach your analysis of a crash. Although the discussion merely touches on the subject of crash dump analysis, it should help you determine which product crashed and with what error condition. At the very least, you should be able to formulate a more precise description of the crash conditions, such as

> I'm getting an error 26 from my version 4.1 Mail System Service

rather than

My system is crashing

This information is extremely useful in the event that you need further assistance from Technical Support.

Where to Start When Your System Crashes

Your system provides valuable clues you can use to determine the cause of a crash. When your system crashes and reboots, 8 bytes of crash information are displayed on the screen (workstations only) and they are logged in *Log.sys*.

Upon a crash, the system displays a message of the following form:

CRASH STATUS (ERC X.) xx xx xx xx xx xx xx xx

where

X.	Is the status code the system crashed with in decimal notation.
xx	Is a crash status word. The leftmost status word is the crash status code in hexadecimal notation. The next word normally is the identification number of the process that was scheduled to run. (The meanings of the crash status words are described in greater detail in "The System Error Buffer," later in this section.)

For your convenience, this crash information is displayed again several times during the reboot process.

You can note down the eight crash status words at this time, or you can use the Executive PLog command to obtain this same information. If you use PLog, the eight crash status words contained in the most recent PLog entry are the words you need to note. (You can also examine the system error buffer. For details, see "System Error Buffer.")

To analyze the crash status on a shared resource processor, the shared resource processor boards have to be configured so that they will reboot. Once the system has rebooted, you can examine the crash status words by using the ClusterView utility. If a shared resource processor crashes but is not configured to reboot, the LEDs on the back panel display the error code. For details on how to interpret the LED display, see the *CTOS Status Codes Reference Manual*. For information on configuring your shared resource processor, see Appendix F in this guide as well as the *CTOS System Administration Guide*.

Using the Debug File Command

If the system memory image was dumped to a disk file, you can use the Debug File command to examine the file. In the Executive, type **Debug File** to display the command form. Fill out the form by entering the name of your crash dump file:

```
Debug File
 File Name           CrashDumpFileName
 [Write?]            ________________________________
 [Image Mode?]       ________________________________
 [Symbol File]       ________________________________
 [Script File]       ________________________________
 [Output File]       ________________________________
```

Then press **GO**. When the **Debug File** prompt is displayed, you can start examining your crash dump file. Following are suggestions for starting your investigation.

Displaying All Processes

To display all the application and system processes in the system when the crash occurred, you can use the **CODE-S** command. (For details on **CODE-S**, see “Displaying Process and Exchange Status: CODE-S” in Section 7, “Display Commands.”) If several pages of information are displayed for the Processes display (first item shown by **CODE-S**), it is very likely that the system did not dump to your crash dump file. This can occur, for example, if your file is not large enough to contain the entire dump. (Guidelines for sizing a crash dump file are included in Appendix G, “Extended Crash Dump Process.”)

Normally when you use the CODE-S command, eight to thirty entries are shown. An example of the Processes display for protected mode is shown in Figure 11–1. (An example of this display for a real mode system is shown in “Displaying Process and Exchange Status: CODE-S” in Section 7, “Display Commands.”)

Figure 11–1. CODE-S Processes Display (Protected Mode)

```
Processes

id oPcb  cs     ip   st pr tss  ldt   ss   sp    ds  exch sgU  user partition
00 6BBA 0298:08B2 C0 02 0880 0000 04C0:99BE 04C0 0007 0000 0000
01 6BCC 0298:08B2 C0 01 0890 0000 04C0:9A70 04C0 000D 0000 0000
02 6BDE 0298:08B2 C0 04 08A0 0000 04C0:9B6C 04C0 000E 0000 0000
03 6BF0 0298:08B2 C0 05 08B0 0000 04C0:9D62 04C0 0010 0000 0000
04 6C02 0298:08B2 C0 06 08C0 0000 04C0:9E5C 04C0 0012 0000 0000
05 6C14 0298:08B2 C0 08 08D0 0000 04C0:9F5C 04C0 0013 0000 0000
06 6C26 0298:08B2 C0 07 08E0 0000 04C0:A0F2 04C0 0014 0000 0000
07 6C38 0298:08B2 C0 04 08F0 0000 0D78:03BE 0D78 001B 0000 0000
08 6C4A 0298:08B2 C0 03 0900 0000 0D78:7D22 0D78 0000 0000 0000
09 6C5C 0298:08B2 C0 06 0910 0000 12C0:FFFA 12C0 001C 0000 0002 Vdm_Ch 2.0
0A 6C6E 0298:08B2 80 14 0920 1498 01A4:96F2 01A4 001E 1190 0003 Primary
0B 6C80 0298:08B2 80 80 0930 15B0 045C:FEB4 045C 0022 1598 0005 CM01
0C 6C92 0298:08B2 80 13 0940 1498 01A4:887A 01A4 0023 1190 0003 Primary
0D 6CA4 0298:08B2 80 0C 0950 1498 01A4:8A96 01A4 0025 1190 0003 Primary
0E 6CB6 0298:08B2 80 7F 0960 15B0 045C:BBD5 045C 0027 1598 0005 CM01
0F 6CC8 F168:001D 80 78 0970 0001 F58B:0BAC F58B 002B 1650 0006 CM02
47 70B8 0318:0018 C1 FF 0CE0 0000 04C0:97DC 04C0 0000 0000 0000

Run Queue |47|
```

The process identification numbers for all processes are contained in the leftmost column of the Processes display.

Normally, the identification number of the process that caused the crash is the second crash status word. (See “Where to Start When Your System Crashes.”) However, the faulting process also can be associated with one of the processes in the Run Queue immediately following the Processes display. When examining processes in the Run Queue, you should start with the leftmost process number in the queue, then proceed to the right. To examine a process, first set PR to that process, as described next.

Setting PR

To examine the process you suspect is in question, set the Debugger's internal register PR to that process.

Type **PR** (press **RIGHT ARROW**)

The process number of the current process is displayed. To the right of this number, type the process number [or the protected mode Task State Segment (TSS) selector] of the process you want to examine. Then press **RETURN**. (Setting PR is also discussed in Section 6, "Working With Registers.")

Loading the Symbol File

After setting PR, you can load the symbol file for the process. Loading symbols allows you to specify public symbols rather than logical addresses for given locations.

For Local Descriptor Table (LDT) based programs and protected mode operating system tasks, the Debugger can locate the load offset for you. Just type the symbol file name within single quotation marks and press **CODE-F**, as shown below:

'SymbolFileName' (press **CODE-F**)

If loading symbols is successful, the **Debug File** prompt appears.

For the following programs, you must calculate the load offset:

- Global Descriptor Table (GDT) based programs (such as system-common procedures) running on operating systems prior to CTOS III.
- RMOS programs (real mode programs running on protected mode operating systems)

Instead of using **CODE-F** to load symbol files, you can have the symbol files loaded automatically. See the discussion on the Resource Librarian command in Section 4, "General Purpose Functions and Features."

Calculating the Load Offset

To arrive at the correct offset, follow the procedures below:

1. Enter 0 as the selector (or real mode segment) value preceding the symbol file name as follows:

   Type **0**, *'SymbolFileName'* (press **CODE-F**)

2. Type a symbol from the data segment (DS) followed by the equals sign (=). For example, you could type

   sbVerrun=

   sbVerrun is the symbolic address of the operating system version string, which occurs in most programs.

   Debug File responds by displaying an address of the form SA:RA, for example

   28:0FB40

   With this response, if the program were loaded at offset 0, DS would be the twenty-eighth selector.

3. To obtain the real DS value,

   type **DS** (press **RIGHT ARROW**)

4. The Debugger displays the DS, such as

   1638

5. Then, load the symbol file at the offset based on the difference between the values of the real DS obtained in steps 3 and 4 and the DS obtained in step 2. For this example, you would type

   [1638-28], *'SymbolFileName'*
   (press **CODE-F**)

6. If the symbol file is loaded successfully, the **Debug File** prompt is displayed.

Verifying the Operating System Symbol File

If you loaded the operating system symbol file, you can verify that it is the correct file by checking the operating system version string. This string is typically displayed on the top line of the screen by applications such as the Executive.

If, for example, Version 9.9 of a real mode operating system is installed on your standalone workstation configuration, the version string is

tlClstrLfsMp 9.9

To verify the version string, you can use the **CODE-D** command. You would type

10.,sbVerrun (press **CODE-D**)

If you loaded the correct symbol file, **Debug File** should display 10 bytes followed by the ASCII characters in the string, as shown below:

```
1AD2:1DDC 09 6E 53 74 6E 64 2D 31 2E 30  nStnd-1.0
```

Note that because the version string is an sb-type string, the string length (9) is contained in the first byte.

If, instead of the version number, garbage is shown for the ASCII characters, the symbol file is not the one created when the operating system was linked. Although it is possible that a different symbol file can successfully dump the version string, this test is accurate in most cases.

Using the Task Register and System Error Buffer

With the operating system symbol file loaded, you can obtain information about the crash by examining the Task Register (protected mode only) and the system error buffer.

Task Register

If you know that a protected mode program called the Crash operation, you can identify the program's Task State Segment (TSS) by typing the symbolic address of the Task Register (TR) as follows:

Type **CrashTr** (press **RIGHT ARROW**)

Debug File displays the TSS of the program that called Crash.

Once you have identified the TSS, you can find the user number and, thereby, identify the run file that crashed. (For details, see "Determining Which Run File Crashed.")

System Error Buffer

If you did not write down the 8 hexadecimal crash status words (described in "Where to Start When Your System Crashes"), you can obtain this information by examining the system error buffer, SysErrorBuf. This buffer is an array of the eight words of crash information that are logged to the system Log file as part of the crash message.

To examine this buffer,

type **SysErrorBuf** (press **RIGHT ARROW**)

The first word of SysErrorBuf is displayed. This word is the error code in hexadecimal that the system crashed with. (To display its decimal equivalent, you can press **CODE-R**. Decimal output is followed with a decimal point.) If the error code is not the one you last crashed with, the system did not dump its memory contents to this file. This can occur if there are file protection problems. To correct this situation, you can use the Executive Set Protection command, then crash the system again. (For details on Set Protection, see the *CTOS Executive Reference Manual.*) This can also occur if the system is not configured properly.

To display the second word of SysErrorBuf,

press **DOWN ARROW**

This word is the number of the process that was scheduled to run (but may not have been running as explained next) when the crash occurred.

The operating system executes code segments either in processes or interrupt handlers, but it does not assign process numbers to the interrupt handlers. If, by chance, a process is executing and an interrupt occurs whose handler crashes the system on a real mode operating system, the process number is stored as the second word of SysErrorBuf. In the same situation on protected mode systems, the TSS of the interrupt handler is stored in this word.

On real mode operating systems only, with an operating symbol file loaded, you can check for this by typing

fProcess (Press **LEFT ARROW**)

If the value displayed is 0, an interrupt handler was active. Otherwise, a 1 is displayed, indicating that the process really was running. If a symbol file is not loaded, you can only assume that the process was running until future crashes show a pattern that indicates otherwise.

The seventh and eighth words of SysErrorBuf contain the address of the instruction following the one that caused the crash. (The values of the third through sixth word of SysErrorBuf depend on the error code. For details, see the *CTOS Status Codes Reference Manual.*)

Determining Which Run File Crashed

There are two methods you can use to determine which run file crashed. These are described next. Note, however, that if the source of the crash is an interrupt handler, you can skip this discussion because the partition information you would use does not apply to interrupt handlers.

With the User Number

By determining the user number of the process that crashed, you can find out which program was running in the partition.

You can look up the user number if you know the TSS or the process identification number. (Methods of obtaining the TSS and the process number are described in "Using the Task Register and System Error Buffer.")

To look up the user number, use **CODE-S**. Examine the Processes display. (See the example in Figure 11–2.) In the display, there is a column for the process identification number, the Task State Segment selector (protected mode), and the user number. These are represented by the column headings id, tss, and user, respectively, in Figure 11–2. You can look up either the TSS or the id in question. Then locate the corresponding user number on the same line as the TSS or the id.

Figure 11–2. CODE-S Processes Display (Protected Mode)

```
Processes

id oPcb  cs        ip    st pr tss  ldt   ss    sp    ds   exch sgU  user partition
00 6BBA 0298:08B2 C0 02 0880 0000 04C0:99BE 04C0 0007 0000 0000
01 6BCC 0298:08B2 C0 01 0890 0000 04C0:9A70 04C0 000D 0000 0000
02 6BDE 0298:08B2 C0 04 08A0 0000 04C0:9B6C 04C0 000E 0000 0000
03 6BF0 0298:08B2 C0 05 08B0 0000 04C0:9D62 04C0 0010 0000 0000
04 6C02 0298:08B2 C0 06 08C0 0000 04C0:9E5C 04C0 0012 0000 0000
05 6C14 0298:08B2 C0 08 08D0 0000 04C0:9F5C 04C0 0013 0000 0000
06 6C26 0298:08B2 C0 07 08E0 0000 04C0:A0F2 04C0 0014 0000 0000
07 6C38 0298:08B2 C0 04 08F0 0000 0D78:03BE 0D78 001B 0000 0000
08 6C4A 0298:08B2 C0 03 0900 0000 0D78:7D22 0D78 0000 0000 0000
09 6C5C 0298:08B2 C0 06 0910 0000 12C0:FFFA 12C0 001C 0000 0002 Vdm_ch 2.0
0A 6C6E 0298:08B2 80 14 0920 1498 01A4:96F2 01A4 001E 1190 0003 Primary
0B 6C80 0298:08B2 80 80 0930 15B0 045C:FEB4 045C 0022 1598 0005 CM01
0C 6C92 0298:08B2 80 13 0940 1498 01A4:887A 01A4 0023 1190 0003 Primary
0D 6CA4 0298:08B2 80 0C 0950 1498 01A4:8A96 01A4 0025 1190 0003 Primary
0E 6CB6 0298:08B2 80 7F 0960 15B0 045C:BBD5 045C 0027 1598 0005 CM01
0F 6CC8 F168:001D 80 78 0970 0001 F58B:0BAC F58B 002B 1650 0006 CM02
47 70B8 0318:0018 C1 FF 0CE0 0000 04C0:97DC 04C0 0000 0000 0000

Run Queue |47|
```

Once you have obtained the user number, you can identify the partition handle. On a workstation, the partition handle is the same as the user number. On a shared resource processor, however, the partition handle is represented by the lower 10 bits of the user number. (A user number assigned by a workstation operating system is 16 bits, but only the lower 10 bits are significant; the upper 6 bits are always zeros. A user number assigned by a shared resource processor utilizes all 16 bits; the upper 6 bits identify the position of the processor board within the shared resource processor where the user number was assigned.) If, for example, the user number is 0802, the partition handle is 2 and the slot is 72. A parition handle of 0 always indicates an operating system process.

With the partition handle, you can use the Executive Partition Status command to determine the name of the crashed file. (For details on Partition Status, see the *CTOS Executive Reference Manual.*)

With the User Structure

For protected mode, there is an alternate method of obtaining the run file name. This method also requires using the **CODE-S** command. In the Processes display there is a column (sgU) for the User (U) Structure. (See the example in Figure 11–2.) The U Structure identifies the system structures, such as the Partition Configuration Block and the Extended Partition Descriptor, for the user number shown on the same line in the Processes display.

If the run file in question is not that of a system process (that is, if sgU is not the value 0), the current run file name is contained within the first 79 bytes of the U structure.

You can use **CODE-D** to display some of the bytes starting at the first byte of the U structure. (For details on **CODE-D**, see “Displaying the Contents of Memory: CODE-D” in Section 5, “Examining and Changing Memory Contents.”)

For example, to examine the first 20 bytes of Sgu for process 0D, you would first set PR to 0D. Then, you would type

20, 1190:0(press **CODE-D**)

where 1190 is the value of the U structure.

A typical response is

```
1190:0  08 00 00 00 00 01 00 12 5B 53 79 73 5D 3C 73 ←  ↑  →  [Sys]<s
1190:10 79 73 3E 45 78 65 63 2E 52 75 6E 52 75 6E 2E 72 ys>Exec.RunRun.r
```

In this example, the run file is *Exec.Run.*

Determining the Run File Version Number

To obtain the version of the run file that crashed, use the **Version** command, specifying the name of the run file that crashed. (For details on the **Version** command, see the *CTOS Executive Reference Manual.*)

Source of Crash

Your action will be different depending on the source of a crash. The paragraphs that follow describe what you should do in the cases where the crash source is a system service, an application, an interrupt handler, or the operating system.

System Service or Application

If the source of the crash is a system service or an application, you can report the problem to the party responsible for supporting that software product.

Interrupt Handler or Operating System

Interrupt Handler

If an interrupt handler crashed, contact Unisys Data Systems Division, San Jose Product Support.

Idle Process

The operating system idle process can crash as a result of malfunctioning hardware or a bad checksum. This process is shown on the last line in a **CODE-S** Processes display. (See Figure 11–3.) It can also be identified by having the lowest (numerically highest) priority (FF).

Figure 11–3. CODE-S Processes Display (Protected Mode)

```
Processes

id oPcb  cs        ip   st pr tss  ldt  ss        sp   ds   exch sgU  user partition
00 6BBA 0298:08B2 C0 02 0880 0000 04C0:99BE 04C0 0007 0000 0000
01 6BCC 0298:08B2 C0 01 0890 0000 04C0:9A70 04C0 000D 0000 0000
02 6BDE 0298:08B2 C0 04 08A0 0000 04C0:9B6C 04C0 000E 0000 0000
03 6BF0 0298:08B2 C0 05 08B0 0000 04C0:9D62 04C0 0010 0000 0000
04 6C02 0298:08B2 C0 06 08C0 0000 04C0:9E5C 04C0 0012 0000 0000
05 6C14 0298:08B2 C0 08 08D0 0000 04C0:9F5C 04C0 0013 0000 0000
06 6C26 0298:08B2 C0 07 08E0 0000 04C0:A0F2 04C0 0014 0000 0000
07 6C38 0298:08B2 C0 04 08F0 0000 0D78:03BE 0D78 001B 0000 0000
08 6C4A 0298:08B2 C0 03 0900 0000 0D78:7D22 0D78 0000 0000 0000
09 6C5C 0298:08B2 C0 06 0910 0000 12C0:FFFA 12C0 001C 0000 0002 Vdm_Ch 2.0
0A 6C6E 0298:08B2 80 14 0920 1498 01A4:96F2 01A4 001E 1190 0003 Primary
0B 6C80 0298:08B2 80 80 0930 15B0 045C:FEB4 045C 0022 1598 0005 CM01
0C 6C92 0298:08B2 80 13 0940 1498 01A4:887A 01A4 0023 1190 0003 Primary
0D 6CA4 0298:08B2 80 0C 0950 1498 01A4:8A96 01A4 0025 1190 0003 Primary
0E 6CB6 0298:08B2 80 7F 0960 15B0 045C:BBD5 045C 0027 1598 0005 CM01
0F 6CC8 F168:001D 80 78 0970 0001 F58B:0BAC F58B 002B 1650 0006 CM02
47 70B8 0318:0018 C1 FF 0CE0 0000 04C0:97DC 04C0 0000 0000 0000

Run Queue |47|
```

The most common reason for the idle process to crash is that malfunctioning hardware has generated stray interrupts or other problems that the operating system is unable to handle. Fixing the hardware resolves this problem.

On CTOS I systems, the idle process checksums the operating system code. If the process calculates a bad checksum, some other process or interrupt handler overwrote part of the operating system code. As a result, the idle process calls the Crash operation with error code 91 ("Operating system checksum error"). Because this error is difficult to analyze, you should contact Technical Support.

Termination Process

Occasionally the operating system is unable to recover from an error when loading a program into a partition. For example, a crash can occur when a nonexistent exit run file is specified and the application then exits. When the operating system finds it cannot run anything in the primary partition, the termination process calls the Crash operation with an appropriate error code. On a shared resource processor, a message is logged in Log.sys and the partition is removed. The system does not crash. You can examine the message by using the Plog command described earlier.

You can determine which user number is at fault by looking for gaps in the numbering sequence for the user numbers displayed by **CODE-S**. On multipartition (real mode 9.10) systems user number 1 may be missing, whereas any number (except 0 or 1) may be missing on protected mode or CTOS/XE systems. Usually the missing user number is the one at fault.

Error Code 22, 26, or 28

An operating system crash with error code 22, 26, or 28 can occur for a number of reasons.

Because system code on CTOS II and CTOS III is not checksummed by the idle process, if the code is overwritten by a system service or application, the system may crash with error code 22, 26, or 28 rather than with error code 91 ("Bad checksum").

Other reasons for the occurrence of these error codes are corrupted pointers as the result of overwritten data areas and hardware-related problems. If you detect any of these error codes, you should contact Unisys Data Systems Division, San Jose Product Support.

Appendix A
Status Messages

Status Messages

The error messages that the Debugger displays are shown below in boldface type. The explanation of each message appears in regular type.

Address must not be in an overlay

You cannot modify code in an overlay.

Breakpoint already set

A previous **CODE-B** command already set a breakpoint at the specified address.

Cannot proceed

You cannot invoke **CODE-P** to resume a process that is already running.

Expected numeric parameter not found

You must use a numeric parameter with this command.

Expected parameter not found

You must use a string parameter with this command.

Expected parameter(s) not found

The parameters of the specified command are not of the correct type or number.

Nonexistent memory

No physical memory exists at the specified address.

No such breakpoint

No breakpoint has been set at the address given in the **CODE-C** command.

No such command

That command does not exist. (For a list of all the Debugger commands, refer to the command summary in the quick reference accompanying this guide or to the online help file available with protected mode Debuggers only.)

Not a symbol file

The file name parameter in the **CODE-F** command is not the name of a symbol file. Check the spelling.

Not allowed when interrupts are disabled

The command in question is not available after a CODE-I breakpoint has been taken, or when interrupts are disabled.

Not enough parameters

You must enter the command again with the correct number of parameters.

Not implemented

The specified command is not implemented in the Debugger.

PatchArea offset too large

The offset in PatchArea must not exceed 49 bytes.

Pattern not found

The specified pattern was not found in the range of addresses given as parameters of the **CODE-O** command.

Radix must be between 2 and 16

The parameter of the **CODE-R** command must be in the range from 2 to 16 (decimal), inclusive.

Segmented address parameter not found

You must use an address parameter with this command.

System error while opening a symbol file

A file system error occurred when the Debugger tried to open the symbol file. You should verify that the file name is spelled correctly.

Too many breakpoints

The Debugger permits only 16 breakpoints at one time.

Too many open symbol files

You cannot open another symbol file because the limit on the number of open symbol files has been reached. You should close a symbol file.

Too many parameters

You must enter the command again with the correct number of parameters.

Appendix B
Shared Resource Processor Debugging

Shared Resource Processor Debugging

Using the Debugger on a shared resource processor is somewhat different from using it on a workstation.

A workstation comes equipped with a keyboard and video screen, both of which are essential for using the Debugger. A shared resource processor does not. To use a keyboard and screen with a shared resource processor, you need a workstation or terminal as well as additional utilities.

Furthermore, there are differences in the way you configure the Debugger for a shared resource processor. Unlike a workstation, a shared resource processor has multiple processor boards, each with its own memory. Therefore, you need to configure the Debugger for each board you want to debug.

When debugging through the Debugger port, there are differences in the way you access individual shared resource processor boards. Because TP, CP, and GP boards have RS-232-C ports, these boards may be accessed directly. All other types of shared resource processor boards, such as FPs, must be accessed indirectly through either a TP, CP, or a GP board.

All of these issues, as well as specific information about shared resource processor debugging, are discussed in this appendix.

The preferred method of debugging on the shared resource processor is to use Administrator ClusterView. This facility allows you to execute commands on shared resource processor boards whereas typically you execute them from your workstation. In certain cases, however, debugging operations are video-dependent and Administrator ClusterView does not provide the proper emulation for this activity. You may want to use Basic ATE or a terminal instead.

Note: *If you are using Basic ATE for debugging on a shared resource processor, you may be able to use certain workstation Debugger command keystrokes directly. See the* Basic ATE User's Guide *for details.*

Configuring the Debugger on Your SRP

To use the Debugger at all on your shared resource processor, you first need to configure the shared resource processor so that the Debugger is loaded into the memory of the appropriate boards.

For the shared resource processor Debugger to be loaded, it is essential that the current copy of the Debugger be contained in the Debugger system file. The specific name of this file varies depending on whether a board is real mode or protected mode. At system boot time, a copy of this file is loaded dynamically into the memory of the specified shared resource processor boards. For the name of the system file, see your release documentation.

You specify which boards are to contain a copy of the Debugger by editing the shared resource processor system configuration files. For each keyswitch position on a shared resource processor, there is a configuration file with a name of the form.

SrpConfig.*k*.sys

where *k* is the keyswitch position as follows: M (manual), R (remote), or N (normal).

Additional information on shared resource processor configuration can be found in your operating system release documentation as well as in the *CTOS System Administration Guide.*

For each processor board on which you want the Debugger loaded, you need to include the following line. Enter the line in the processor section of the configuration file for that keyswitch position:

```
:LoadDebugger: Yes
```

To debug with Administrator ClusterView, this is all that is needed. Then, you can activate the Debugger by pressing **HELP-A** (or **HELP-B**) on a shared resource processor.

To debug with Basic ATE or a terminal connected to a TP, CP, or GP, you must, in addition, include other information in the configuration file, as described in "Using a Debugger Port on any Shared Resource Processor Board," next in this appendix.

Using a Debugger Port on any SRP Board

Because the shared resource processor boards have RS-232-C connectors, you can also access the Debugger on any shared resource processor board directly by using an asynchronous terminal or a workstation that is running Basic ATE. To connect the terminal to a shared resource processor board, you need to construct a null modem cable. The configuration for the null modem cable is shown in Figure B–1.

Figure B–1. Null Modem Cable Configuration

```
25 Pin RS-232-C         25 Pin RS-232-C
Male Connector          Male Connector

Pin 2  --------------   Pin 3
Pin 3  --------------   Pin 2
Pin 7  --------------   Pin 7
Pin 6  ---|
Pin 8  ---|----------   Pin 20
Pin 20 -----------|--   Pin 6
                  |--   Pin 8
```

In addition, you need to add an entry to the processor section of the configuration file to define the RS-232-C line to be used by the Debugger. This entry has the form:

```
:DebugPort: Yes
```

In this case, the Debugger defaults to those parameters used by Basic ATE. For a list of the default values, see your release documentation. The actual port defaulted to is different for each type of processor.

There is another way to use the entry in the configuration file that allows specifying different communication defaults. It also provides access to the Debugger on an FP, DP, or SP board by using an RS-232-C port on a CP, TP, or GP board. This entry has the form:

```
:DebugPort:    (Speed = baud rate,
                Parity = parity value,
                Stopbits = number,
                Charbits = number,
                Modem = Yes or No,
                Port = alphanumeric,
                Processor = xPnn)
```

Using this form allows specifying different communication defaults. It also allows access to the Debugger on an FP, DP, or SP board by using an RS-232-C port on a CP, TP, or GP board.

The parameters and their values are as follows:

Speed

Acceptable baud rates include 50, 75, 110, 134, 150, 200, 300, 600, 1200, 2400, 4800, 9600, 19200. Basic ATE defaults to 9600.

Parity

Acceptable parity values include none, even, odd, 0, or 1. Basic ATE defaults to 0. Since 0 or 1 forces the high order bit to be 0 or 1, you cannot specify a parity of 0 or 1 if you specify 8 character bits in the "Charbits" parameter, described below.

Stopbits

Configurable as 1 or 2.

Charbits

You can specify 5, 6, 7, or 8. You need to configure this depending on the terminal you are using. The default is 7 bits.

Modem

You can specify Yes or No. This option is supported in the following cases only:

On a TP: when Debugger Port is set to channel 4, 5, 6, 7, 8, or 9
On a CP: when Debugger Port is set to channel 2

Port

Set this parameter as follows:

For a GP: 0 or 1
For GP+CI: 0, 1, 2, 3, 4, 5, 6, or 7
For a CP: 0, 1, or 2
For a TP: 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9

You can use letters (A through J) instead of numbers. Use this if you want to use a port other than the default, which may be in use. This parameter is required for FP, DP, and SP boards.

Processor

This parameter is required for FP and DP boards, and is not supported on GP boards.

For boards that do not have their own RS-232-C ports, you can borrow a port from another processor. For example, you can debug an FP board by connecting Basic ATE to a port on a CP board. You can borrow from a CP or TP board only. An example of an acceptable value would be CP00.

You can enter the above parameters in any order, separated by white space or commas. Here is an example of what an entry may look like:

```
:DebugPort: (Speed = 9600, Parity = 0, Stopbits = 1, Processor =
CP01)
```

For more information about how to enter values in a configuration file, see the *CTOS System Administration Guide*.

You must connect the terminal or workstation (running Basic ATE) to the shared resource processor board using the null modem cable configuration as shown in Figure B–1. If you do not specify the port number in :DebugPort:, the default is CHANNEL 3 (Port = 2) for a CP, CHANNEL 10 (Port = 9) for a TP, and CHANNEL B (Port = 1) for a GP.

Note: *You must use the debug port for debugging with interrupts disabled (see CODE-I). Should you use CODE-I breakpoints on a processor with a cluster, all cluster workstations will time out and not have access to the server.*

Once you have made the necessary connections, you should be communicating with the processor board to which you are connected. To verify that you are, press **CODE-A** once. A Debugger prompt should appear.

Caution

If you are using Basic ATE to access a shared resource processor Debugger, pressing **FINISH** terminates Basic ATE but does not resume the Debugger. Consequently, your process may be halted in the Debugger. To exit the Debugger from DebugPort, first be sure the F10 function key is toggled to 'workstation keyboard for shared resource processor debugging mode' from regular Basic ATE mode. Then, press **GO**. The message "Exiting Debugger" should appear.

Appendix C
Stack Format

What is a Stack?

A *stack* is a region of memory used by a program to save the environment when a procedural call is made and to handle parameter passing. A stack is also used by interrupt and trap handlers to save the context of a process when an interrupt occurs.

This appendix describes stack format and discusses how to use the stack as an aid in debugging your programs.

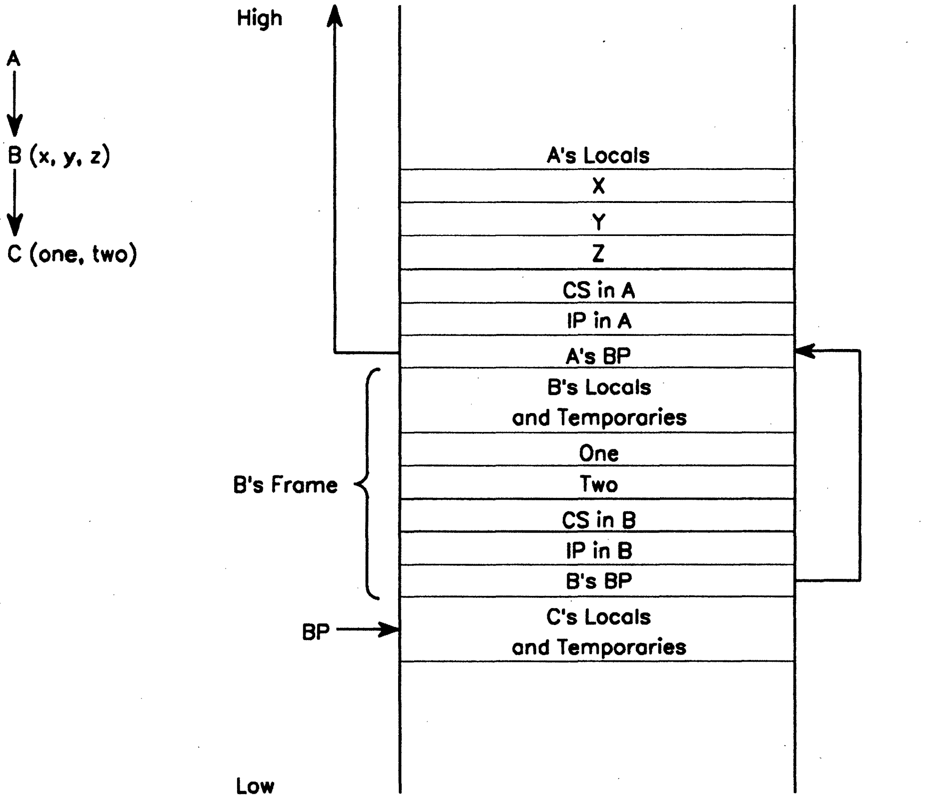
Format of the Stack

The format of the stack is illustrated in Figure C–1. The initial value of the stack pointer (SP) register is at the highest address of the stack. The stack grows downward toward lower addresses as new words are pushed onto it. Each location on the stack is a word. Associated with the stack are stack pointers and stack frames.

A *stack pointer* is an indicator to the top of the stack. Access to the stack is always via a stack pointer. Stack pointers have the format SS:SP. Since the stack grows downward, the value of SP, which points to the top of the stack, decreases as the stack grows.

A *stack frame* is a region of the stack corresponding to the dynamic invocation of a procedure. Stack frames consist of procedural parameters, a return address, a saved-frame pointer, and local variables.

Figure C–1. Format of the Stack



558.C–1

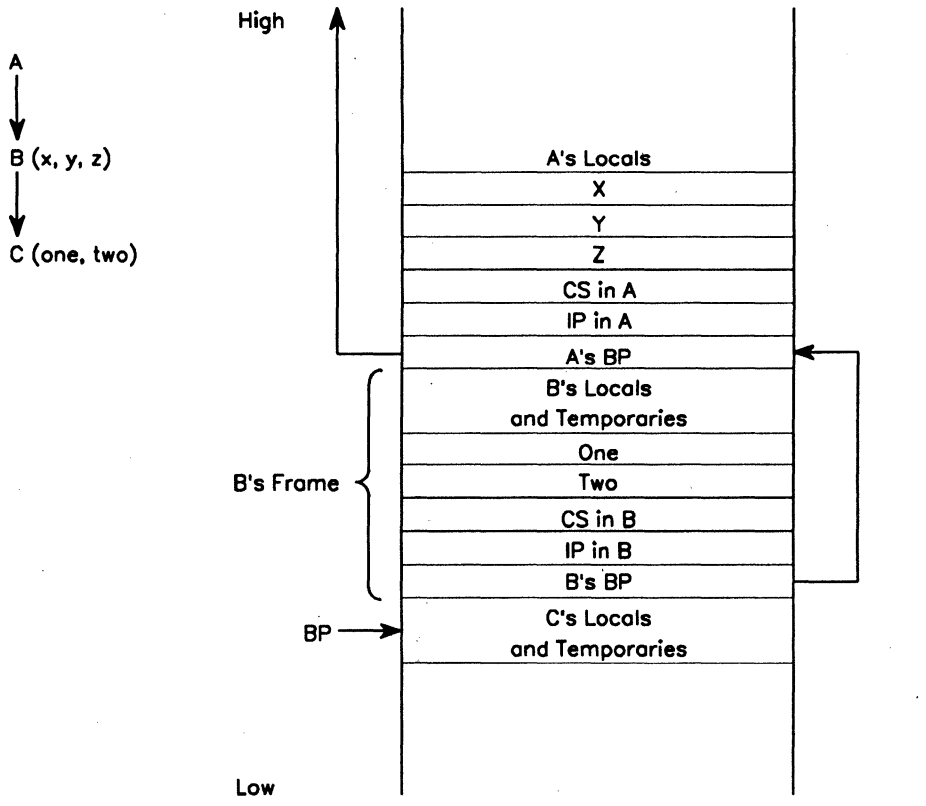
Each stack frame has an associated *base pointer* (BP). The BP is a point of reference used by the called procedure to find the values of passed or local parameters.

The Debugger accesses the stack through a *stack trace*. A stack trace is a display of the program's stack arranged by stack frames. The Debugger command **CODE-T** is used to display the stack.

Interpreting a Stack

The stack format shown in Figure C–2 illustrates two nested procedure calls. Procedure A has called procedure B which, in turn, has called procedure C. For each call there is a stack frame.

Figure C–2. Format of the Stack



558.C–1

Note: *Stack conventions concerning local variables and parameter passing are language specific. This appendix shows the stack as it would appear when using either a Pascal or PL/M compiler. For details, see your language manual.*

The process that created the example stack is as follows:

1. When procedure A calls procedure B, the values of A's local variables are on the stack.
2. The passed parameters X, Y, and Z are pushed in the same order in which they appear in the procedure call.
3. The values of the CS and IP registers are pushed. These values indicate the point at which execution in A will continue after the return.
4. Finally, the value of the BP register is pushed.

While procedure A was executing, before it called procedure B, the BP register pointed to the location immediately above A's local variables.

When A calls B, that value of the BP register must be saved for the return. It is the last item pushed onto the stack. In Figure C–2, it is denoted as A's BP.

After several calls, a chain of BP values has been created that marks the various frames. It is possible to trace back through the stack by following this chain from one BP to the previous one, and so on.

The Virtual Code Management facility does such a stack trace when an overlay is discarded. Virtual Code Management follows the chain of BP values and, by reference from them, corrects the return CS:IP values of any discarded procedures to point to the Overlay Manager.

The Debugger also does a stack trace when you give a **CODE-T** command.

Using the Debugger with the Stack

You can use the Debugger to look at the instructions generated by a compiler to immediately precede and follow a procedure call. The *prologue* is the first instruction of the procedure that saves BP and sets SP. The *epilogue* is the last instruction of the procedure that restores BP and SP. Figure C–3 illustrates a prologue and epilogue.

In the prologue shown in Figure C–3, the following process is illustrated:

- The value of BP is pushed onto the stack.
- Then BP is set equal to SP, thereby setting up a pointer to the current frame.
- The SUB SP,n instruction subtracts the number of bytes (n) of stack space for local variables from the value of SP.

The result of this process is that the correct top-of-stack position is maintained after the called procedure's local and temporary variables have been placed on the stack.

In the epilogue shown in Figure C–3, the following process is illustrated:

- The stack pointer is set equal to the base pointer, thereby eliminating the local and temporary variables from the stack.
- The value of the previous BP (the next location) is popped so that BP points to its previous location in the BP chain.
- The RET m instruction pops and uses the saved CS and IP values to set CS:IP back to the proper location in the calling code segment. It also pops m bytes of the passed variables by setting SP to SP-m (return to CS:IP). This process leaves SP at the low end of the previous procedure's local and temporary variables.

Figure C–3. Procedure Prologue and Epilogue

```
PUSH BP      }
MOV BP,SP    }    Prologue
SUB SP,n     }
.
.
.
MOV SP,BP    }
POP BP       }    Epilogue
RET m        }
```

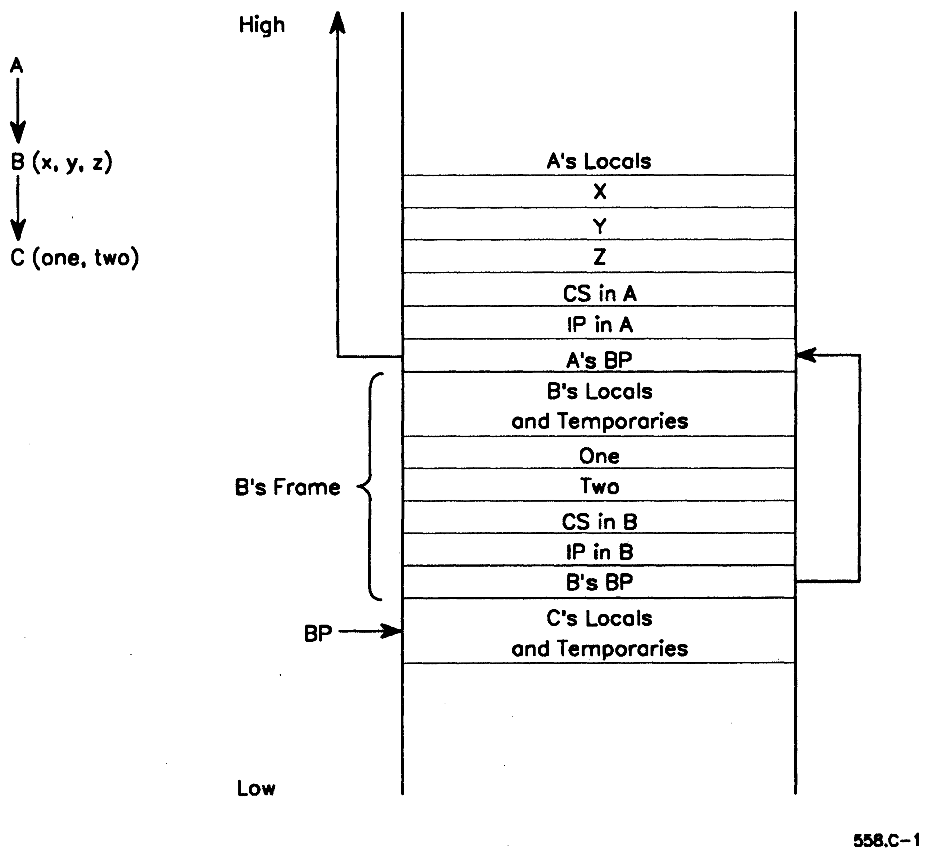
Debugging Using the Stack Trace

The **CODE-T** command displays a stack trace. Details of this command and its usage are given in Section 7, "Display Commands." Shown below are the first few lines of that stack trace.

```
0   3108   12D2:14A   (3,3,4F1F,3113)
1   3116   581F:95    (173C,1,4F1F,312A,4F1F,312E)
2   3130   5EC8:0C4   (4F1F,173C,4F1F,314D)
```

The first column is the level number of the stack frame described by that line. (See Figure C–4.) You can see by comparing the trace to Figure C–4 that the Debugger reads toward the beginning of the frame (from lower to higher addresses) and displays what it finds in each category of items pushed.

Figure C–4. Format of the Stack



The second column contains the value of BP for the frame, and the second column collectively is the BP chain. For example, the value in location 3108 in this example would be 3116, and the value in location 3116 would be 3130.

The third column contains the value of CS:IP at which execution will continue when control is returned to this procedure.

The last column lists the values of the parameters passed from this procedure to the next. The Debugger estimates the number of passed parameters to a maximum of six. It assumes word values so a double-word or pointer takes two entries in this list.

Normally, the Debugger lists the parameters in the order in which they were pushed. If, however, the Debugger overestimated the number of parameters, it reaches up into the local area of the calling procedure's stack and presents nonsignificant values at the left end of the list.

When you are debugging, if you know the number of parameters that should have been passed, you can examine the last column, taking double words and pointers into account, starting at an intermediate point and proceeding to the right end of the parameter list.

Estimating Stack Size

To estimate the needed stack size and avoid problems, you can run the program under the Debugger and set a breakpoint at the end of execution or at another convenient point after the stack has just reached its largest requirement.

Since the stack is initialized to zeroes, you can now check to see how much of the low part of the stack is still zeroes in order to find the maximum requirement. Allow another 64 bytes for interrupt handlers plus 448 bytes for making requests (512 bytes total), and reduce the stack size accordingly.

Correcting Stack Overflow

If your program's stack requirements exceed the stack size, the stack can overwrite whatever precedes it in the link map. You can detect this by abnormal program behavior.

If this happens, increase the stack to the maximum allowed within the limits of your data segment. Then relink the program and see if it runs. If it runs, you can then begin reducing stack size according to the process for estimating given above.

Interrupts and the Stack

If interrupts are enabled, the interrupt handlers use the stack as defined by SS:SP. Therefore, you should be careful to never put data such as temporary variables in the stack segment at a memory address lower than SS:SP.

Appendix D
Debugger Tips

Debugger Tips Introduction

This appendix contains other practical examples of ways to use the Debugger. It is assumed that you are at least familiar with the DisplayFile program described in Section 3, "Debugging Session." Portions of that program are used as examples in this appendix to illustrate other debugging techniques not covered in Section 3.

If you debugged programs without ever using a debugging tool, you probably added extra output statements to modules to see if the modules worked in the way you expected. The Debugger eliminates the need to add these extra statements, because you can check the parameters passed to procedures and the error codes returned. These particular error checking functions are the most common reasons for using the Debugger.

If a procedure calls a CTOS operation that returns an error code, the procedure should be written as a function. By doing so, you can check the error returned in the AX register. In this way, you can focus on those procedures that return nonzero error codes.

As you develop skill in examining the stack, you will be able to use the Debugger to troubleshoot faulty procedures in greater detail. Two exercises in this appendix provide you with hands-on experience and a logical approach to looking at what is on the stack.

Remember that corrections you make to your program while in the Debugger are only temporary patches. This is because you are only correcting a copy of the run file in memory, not the run file itself that is stored on disk. To make permanent corrections to the file copy of your run file, you can use the **Debug File** utility. However, you will not be able to set and execute breakpoints. (For details on Debug File, see Section 11, "The Executive Command: Debug File.") Eventually, you will need to edit your source file and relink.

Viewing Code

In the Debugger, you can view the code in your program and in any libraries linked with it. Unless you are explicitly debugging the operating system itself, however, you should not try to view the code in operating system calls (that is, requests, Kernel primitives, and system-common procedures).

Setting Breakpoints

You can set breakpoints anywhere in your program except at the level of CTOS operations. CTOS operations include the operating system calls as well as the operating system library routines. All of the CTOS operations are listed in the *CTOS Procedural Interface Reference Manual.*

As an example, the DisplayFile program described in Section 3, "Debugging Session," contains a call to the operating system library procedure, OpenByteStream. You can set a breakpoint at the call to OpenByteStream; for example,

```
*ENTGQQ+2A▲   CALL OPENBYTESTREAM   .^b
```

By breaking at this point, all instructions are executed up through the instruction that pushes the last parameter to OpenByteStream onto the stack. The actual call to OpenByteStream is not executed.

If, however, you were to set a breakpoint as shown

OpenByteStream ^b

you would be setting the break at the first instruction within the OpenByteStream code. It should not be necessary to examine the code in this library procedure because the code is designed to work correctly for you. You do need to be concerned, however, with the information your program passes to this procedure.

Caution

Setting breakpoints at any instruction in a request, Kernel primitive, or system-common procedure can cause unpredictable results in your program or other programs in the system.

If you need to locate a CTOS operation and you cannot determine how to arrive at its address, you can write a PUBLIC function in your program that just calls that operation. For example,

```
FUNCTION WriteAByte (pbsOut: Pointer; b: Byte): ErcType
[PUBLIC];
VAR erc : Word;
    BEGIN
        WriteAByte := WriteByte (pbsOut, b);
    END;
```

Then, you can set a breakpoint in the Debugger at the symbolic address of the module you wrote, for example

WriteAByte ^b GO

To be able to use symbols, you will need to load your symbol file first. For details on symbol files, see "Symbol Files" in Section 4, "Basic Debugger Functions and Features."

Note: *In protected mode, symbols will not work using certain memory models, such as the small and compact models available with High C. To use symbols, select a memory model that does not group code segments together. For details, see your language manual.*

Looking at the Stack

The following paragraphs describe two approaches to looking at the stack. Each approach is accompanied by script and supporting comments.

Exercise 1

Program errors are often the result of errors in passing parameters. Because all parameters are pushed onto the stack before a call, you can look at the stack just before a procedure call is executed. The approach to looking at the stack demonstrated by exercise 1 is summarized below.

To view the stack in the Debugger starting at the last parameter pushed, you set a breakpoint at the call, as shown in the following example:

```
*ENTGQQ+2A MARK  CALL OPENBYTESTREAM  .^b  GO
```

Then, you examine the contents of the stack, starting at the address of the stack pointer, as shown below:

```
ss:sp (press RIGHT ARROW)
```

The Debugger displays the last word pushed onto the stack before the call. The word in this case would be the buffer size (last parameter to OpenByteStream).

Then, you can press **DOWN ARROW** to view the other words on the stack.

On the following pages the DisplayFile program you debugged in Section 3 is used again to provide you with hands-on experience in viewing the stack for relevant information. Although there are compiler differences in the way parameters and local variables may appear on a stack, you can generally apply the approach used in this exercise to examine the stack at any time while you are debugging. (For language-specific differences, see your language manual.)

To complete the exercise on the next few pages (as well as other exercises in this appendix), you need the following files for the DisplayFile program:

- *DisplayFile.pas*
- *DisplayFile.sym*
- *DisplayFile.run*

These files are contained in the *<Program>* directory on the diskette packaged with this guide. Copy the files into your working directory. For a description of these files and the DisplayFile program, see Section 3, "Debugging Session."

If you recall, there is a bug in the DisplayFile program. In the Debugger, you need to correct this bug before you can execute the program. Otherwise, the program will terminate with an error. As stated earlier in this appendix, Debugger patches are temporary because you are correcting only the copy of the run file in memory, not the run file on disk. For this reason, you need to correct the DisplayFile run-time error once again. The first 14 Debugger lines of this exercise guide you to the erroneous MOV instruction. You need to correct the instruction before proceeding any further.

Run the DisplayFile program, and enter the Debugger by pressing **CODE-GO**. Then, follow the script and commentary on the next few pages.

Script

Line number	Script
1	***'DisplayFile.sym'^f**
2	***DisplayFile^b** **GO**
	*
3	Exiting Debugger
4	Break at ENTGQQ in process 0D
5	***cs:ip MARK** PUSH BP ↓
6	*ENTGQQ+1 **MARK** MOV BP,SP ↓
7	*ENTGQQ+3 **MARK** SUB SP,4 ↓
8	*ENTGQQ+7 **MARK** MOV AX, 0F0F8 ↓
9	*ENTGQQ+0A **MARK** PUSH DS ↓
10	*ENTGQQ+0B **MARK** PUSH AX ↓
11	*ENTGQQ+0C **MARK** MOV AX, 0FD2D ↓
12	*ENTGQQ+0F **MARK** PUSH DS ↓
13	*ENTGQQ+10 **MARK** PUSH AX ↓
14	*ENTGQQ+11 **MARK** MOV AX,0E **MOV AX,0F** ↓

Comments

Line number	Comment
1	First load the symbol file. type **'DisplayFile.sym'** (press **CODE-F**)
2	Then, set a breakpoint at the beginning of the program. type **DisplayFile** (press **CODE-B**) Execute up to the breakpoint by pressing **GO**.
3	Exiting Debugger
4	Break at ENTGQQ in process 0D
5	To display the next instruction to be executed, type **CS:IP** (press **MARK**)
6 - 13	To advance through the MOV and PUSH instructions on lines 6 through 13 leading up to the erroneous MOV instruction on line 14, press **DOWN ARROW** For an explanation of these instructions, see the tutorial in Section 3, "Debugging Session."
14	Correct the MOV instruction on line 14 as shown in the script. This correction will cause the program to run without generating an error. Then press **DOWN ARROW**

Script

Line number	Script
15	*ENTGQQ+14 **MARK** PUSH AX ↑
16	*ENTGQQ+11 **MARK** MOV AX,0F ↓
17	*ENTGQQ+14 **MARK** PUSH AX ↓
18	*ENTGQQ+15 **MARK** MOV AX, 0FD3C ↓
19	*ENTGQQ+18 **MARK** PUSH DS ↓
20	*ENTGQQ+19 **MARK** PUSH AX ↓
21	*ENTGQQ+1A **MARK** XOR AX,AX ↓
22	*ENTGQQ+1C **MARK** PUSH AX ↓
23	*ENTGQQ+1D **MARK** MOV AX, 6D74 ↓
24	*ENTGQQ+20 **MARK** PUSH AX ↓
25	*ENTGQQ+21 **MARK** MOV AX, 0F136 ↓
26	*ENTGQQ+24 **MARK** PUSH DS ↓
27	*ENTGQQ+25 **MARK** PUSH AX ↓
28	*ENTGQQ+26 **MARK** MOV AX,400 ↓
29	*ENTGQQ+29 **MARK** PUSH AX ↓

Comments

Line number	Comment
15	To view the previous instruction again, press **UP ARROW**
16	The instruction is now correct on line 16.
17 - 29	To view the series of MOV and PUSH instructions on lines 17 through 29 leading up to the call to OpenByteStream, continue pressing **DOWN ARROW** For an explanation of these instructions, see the tutorial in Section 3, "Debugging Session."

Script

Line number	Script
30	`*ENTGQQ+2A` **MARK** `CALL OPENBYTESTREAM .`**^b GO**
31	`Exiting Debugger`
32	`Break at ENTGQQ +2A in process 0D`
33	`*`**`ss:sp`** `→ 0400 ↓`

34	`0DBF2:0F06A → F136 ↓`
	`0DBF2:0F06C → DBF2 ↓`
	`0DBF2:0F06E → 6D74 ↓`
	`0DBF2:0F070 → 0000 ↓`
	`0DBF2:0F072 → FD3C ↓`
	`0DBF2:0F074 → DBF2 ↓`
	`0DBF2:0F076 → 000F ↓`
	`0DBF2:0F078 → FD2D ↓`
	`0DBF2:0F07A → DBF2 ↓`
	`0DBF2:0F07C → F0AB ↓`
	`0DBF2:0F07E → DBF2 ↓`

Comments

Line number **Comment**

30 At the call to OpenByteStream, set a breakpoint as shown, and press **GO**.

31 Exiting Debugger

32 Break at ENTGQQ+2A in process 0D

33 To display the word at the stack pointer,

  type **ss:sp** (press **RIGHT ARROW**)

The Debugger displays 0400, the size of the buffer to OpenByteStream.

  press **DOWN ARROW**

34 Until you see all the parameters shown in the script,

  continue pressing **DOWN ARROW**

(Do not exit the Debugger if you plan to continue with the next example in this appendix.) The meaning of each word on the stack in this example is as follows:

Stack Address				Word Meaning
ss:sp →		0400	↓	sBufferArea
0DBF2:0F06A	→	F136	↓	BufferArea RA
0DBF2:0F06C	→	DBF2	↓	BufferArea SA
0DBF2:0F06E	→	6D74	↓	mode
0DBF2:0F070	→	0000	↓	cbPassword
0DBF2:0F072	→	FD3C	↓	bPassword RA
0DBF2:0F074	→	DBF2	↓	bPassword SA
0DBF2:0F076	→	000F	↓	cbFileSpec
0DBF2:0F078	→	FD2D	↓	bFileSpec RA
0DBF2:0F07A	→	DBF2	↓	bFileSpec SA
0DBF2:0F07C	→	F0F8	↓	BSWA RA
0DBF2:0F07E	→	DBF2	↓	BSWA SA

If you look at the parameters to OpenByteStream (shown below), you will note that the Debugger stack display is in the reverse order:

```
FUNCTION OpenByteStream (
pBSWA                   : Pointer;
pbFileSpec              : Pointer;
cbFileSpec              : Word;
pbPassWord              : Pointer;
cbPassWord              : Word;
mode                    : Word;
pBufferArea             : Pointer;
sBufferArea             : Word):      ErcType;
EXTERN;
```

To help you identify the parameters to a call, note down the PUSH and MOV instructions leading up to the call as you view these instructions in the Debugger using the **DOWN ARROW**. If you have a parallel printer, you can attach it to your workstation and use the **CODE-L** command to print your Debugger output. (For details on CODE-L, see "Printing the Debugger Screen: CODE-L" in Section 7, "Display Commands.")

Then, construct a picture of the stack by associating each parameter with a word displayed on the stack. Figure D–1 is an example of the stack before the call to OpenByteStream.

Note: *Stack conventions concerning local variables and parameter passing are language specific. This guide shows the stack as it would appear when using either a Pascal or PL/M compiler. For details, see your language manual.*

Although the order and size of parameters passed may differ depending on your language compiler, there are still some general clues you can use to identify what is on the stack.

In interpreting the stack, you should be able to recognize the parameters passed by value, such as byte counts and buffer sizes. If one of the parameters is a file specification, for example, you should be able to find the count of bytes in the specification as well as the count of bytes in the password. Every pb/cb pair consists of three PUSH instructions. One instruction pushes the word containing the count of bytes. The other two instructions push the segment address (SA) and the offset (RA) of the pointer to the specification.

Figure D–1. Stack Before Call to OpenByteStream

Parameters		**Stack Contents**	
pBSWA	SP	DS	0DBF2
		AX	0F0F8
pbFileSpec		DS	0DBF2
		AX	0FD2D
cbFileSpec		AX	F
pbPassword		DS	0DBF2
		AX	0FD3C
cbPassword		AX	0
mode		AX	06D74
pBufferArea		DS	0DBF2
		AX	0F136
sBufferArea		AX	00400

If you do not exit the Debugger at this point, you can continue with the next example. Otherwise, to clear the breakpoint you set

press **CODE-C**

To exit the Debugger,

press **GO**

Exercise 2

The following is another exercise in examining the stack. If you just did the last exercise (and are still in the Debugger), you can continue with the next exercise at the point where you left off. Just press **RETURN** and follow the script and commentary on the next few pages.

If, however, you exited the Debugger or did not try exercise 1, you will need to correct the error in the DisplayFile program. To do so, follow the first 14 numbered comments in the previous exercise. For details on the files you need and how to enter the Debugger, see the introductory comments at the beginning of that exercise.

Exercise 2 assumes your familiarity with the DisplayFile program, which is described in detail in Section 3, "Debugging Session." Basically, the program reads the program listing file *DisplayFile.pas* shown in Figure 3–1 and writes this file to the screen. The TypeSub procedure is called to do the reads and writes. To write each byte, TypeSub calls the WriteAByte function. The code to TypeSub and WriteAByte is shown in Figure D–2.

In this exercise, you will execute the DisplayFile program up to the call to WriteAByte. At this point in the program's execution, you will examine the last word pushed onto the stack. The lower byte of this word is the first character to be written to the screen. Follow the script and comments on the next few pages.

Figure D–2. TypeSub and WriteAByte

```
FUNCTION WriteAByte (pbsOut: Pointer; b: Byte): ErcType
[PUBLIC];
VAR erc : Word;
BEGIN
   WriteAByte := WriteByte (pbsOut, b);
END;

PROCEDURE TypeSub (pbsIn, pbsOut: Pointer) [PUBLIC];
VAR b: Byte;
erc : Word;

BEGIN
WHILE TRUE DO

  BEGIN

     erc:= ReadByte (pbsIn, ADS b);
     If erc = ercEOF then
     RETURN;
     If erc <> ercOK then
         FatalError (erc);
     count := count + 1;
     CheckErc (WriteAByte (pbsOut, b));
  END;

END;
```

Script

Line number	Script
15	***TYPESUB MARK** PUSH BP ↓
16	*TYPESUB+1 **MARK** MOV BP, SP ↓
17	*TYPESUB+3 **MARK** SUB BP, 4 ↓
18	*TYPESUB+7 **MARK** PUSH WORD PTR [BP+0C] ↓
19	*TYPESUB+0A **MARK** PUSH WORD PTR [BP+0A] ↓
20	*TYPESUB+0D **MARK** LEA, AX, WORD PTR [BP-2] ↓
21	*TYPESUB+10 **MARK** PUSH DS ↓
22	*TYPESUB+11 **MARK** PUSH AX ↓

Comments

Line number	Comment
1 - 14	Follow the first 14 numbered comments in the previous exercise. Then press **RETURN**, and continue with line 15 below.
15	Locate the first instruction in the TypeSub procedure. Type **TypeSub** (press **MARK**) PUSH BP saves the value of the base pointer (BP). Collectively, this instruction and the instructions on lines 16 and 17 are the stack prologue. (For details on the format of the stack, see Appendix C, "Stack Format.") To view lines 16 and 17, press **DOWN ARROW**
16	MOV BP, SP
17	SUB BP, 4 subtracts 4 bytes from the stack pointer to allocate space for local or temporary variables.
18 - 22	To view the instructions on lines 18 through 22, continue pressing **DOWN ARROW** These instructions push the parameters onto the stack for the call to ReadByte. Note that the parameters are not public variables, so their locations are displayed, for example, [BP-2]. In this case, the variables are a parameter and an address of a local variable. You can identify the local variable [BP-2], because local variables are located at higher stack addresses than BP. Since the stack grows downwards, these locations are expressed negatively relative to BP. The locations of nonlocal parameters, on the other hand, are expressed positively because they are positioned below BP. Press **DOWN ARROW**

Script

Line number	Script
23	*TYPESUB+12 **MARK** CALL READBYTE ↓
24	*TYPESUB+17 **MARK** MOV WORD PTR [BP-4], AX ↓
25	*TYPESUB+1A **MARK** CMP WORD PTR [BP-4], 1 ↓
26	*TYPESUB+1E **MARK** JNE .+4 (TYPESUB+22) ↓
27	*TYPESUB+20 **MARK** JMP .+2A (TYPESUB+4A) ↓
28	*TYPESUB+22 **MARK** CMP WORD PTR [BP-4],0 ↓
29	*TYPESUB+26 **MARK** ME .+0A (TYPESUB+30) ↓
30	*TYPESUB+28 **MARK** PUSH WORD PTR [BP-4] ↓
31	*TYPESUB+2B **MARK** CALL FATALERROR ↓
32	*TYPESUB+30 **MARK** INC WORD PTR [COUNT] ↓

Comments

Line number	Comment
23	This is the call to ReadByte.
24 - 32	To view the instructions on lines 24 through 32, continue pressing **DOWN ARROW** On these lines the error code returned in AX (from ReadByte) is examined. Following are brief descriptions of these lines: On line 24 the error code is moved to the location [BP-4]. On line 25 the value of the error code is compared to 1 (EOF). Line 26 jumps 5 bytes to TYPESUB+22 if the error code value is not the EOF. Otherwise, a jump is made to the end of the TYPESUB procedure on line 27. On line 28 the error code is compared to 0 (ercOK). If the error code is 0, line 29 jumps 10 bytes to TYPESUB+30 (shown on line 32). Line 32 then increments the count variable by 1. Otherwise (if the error code is not 0), line 30 pushes the error code onto the stack. Then, on line 31 FatalError is called.

Script

Line number	Script
33	`*TYPESUB+34` **MARK** `PUSH WORD PTR [BP+8]` ↓
34	`*TYPESUB+37` **MARK** `PUSH WORD PTR [BP+6]` ↓
35	`*TYPESUB+3A` **MARK** `PUSH WORD PTR [BP-2]` ↓
36	`*TYPESUB+3D` **MARK** `CALL WRITEABYTE` **.^b** **GO**
37	`Exiting Debugger`
38	`Break at TYPESUB +3D in process 0D`
39	**`*ss:sp`** `037B` **RETURN**
40	**`*2, ss:sp`** **`^d`**
	`0DBF1:0F072   7B  03                    {¢`

Comments

Line number	Comment
33 - 35	To view the instructions on lines 33 through 35, press **DOWN ARROW** On these lines, the parameters and a local variable are pushed onto the stack for the call to WriteAByte. Lines 33 and 34 push the segment address and offset of the variable pbsOut. On line 35 the byte to be written to the screen is pushed.
36	Set a breakpoint at the call to WriteAByte as shown. Then press **GO**.
37	Exiting Debugger
38	Break at TYPESUB+3D in process 0D
39	To check the word at the stack pointer, type **ss:sp** (press **RIGHT ARROW**) The Debugger displays 037B. Your concern is with the lower byte (7B) of this word. This byte is the first byte to be written to the screen by WriteAByte. Press **RETURN**
40	To display the ASCII characters represented by the word, type **2, ss:sp** (press **CODE-D**) The Debugger displays the ASCII codes and the characters they represent for 2 bytes starting at ss:sp. The lower byte (7B) is the ASCII character, Left bracket ({). Compare this character to the first character in the DisplayFile program listing. (See the program listing shown in Figure 3–1 in Section 3, "Debugging Session.")

Script

Line number	Script
41	`*^c` GO

Comments

Line number	Comment
41	To clear the breakpoint you set, press **CODE-C** To exit the Debugger, press **GO** The DisplayFile program executes to completion, displaying the *DisplayFile.pas* file to the screen. The string 'Finished' appears following the last file character.

String Moves

In the exercise that follows, you will be introduced to assembly language instructions used to move strings of bytes from one location to another.

To work through this exercise, you will need to run the DisplayFile program used and described in Section 3 and in the earlier exercises in this appendix. (For details on the files you need, see exercise 1.)

This time, you will execute the DisplayFile program up to the point at which the string variable 'Finished' is assigned to the public variable done in the Process procedure. (The Process procedure is shown in Figure D–3.) At this point in the Debugger, you will execute MOVSW, a word move instruction, which moves a word of the string to the address done. Then you will examine the ES and DS registers to see what was actually moved.

Figure D–3. Process Procedure

```
PROCEDURE Process [PUBLIC];
BEGIN
     count := 0;
     done := 'Finished';
     TypeSub (ADS BSWA, ADS bsVid);
     CheckErc (WriteBsRecord (ADS bsVid, ADS done, 9,
     ADS junk));
END;
```

To enter the Debugger,

press **CODE-GO**

As stated earlier in this appendix, there is a bug in the program you need to correct before proceeding with this exercise. Otherwise, the program will terminate with an error. To correct the bug, follow the first 14 numbered comments in "Exercise 1," earlier in this appendix. Then proceed with the script and comments on the next few pages.

This page intentionally left blank

Script

Line number	Script
15	***Process MARK** PUSH BP ↓
16	PROCESS+1 **MARK** MOV BP, SP ↓
17	PROCESS+1 **MARK** SUB BP,0 ↓
18	PROCESS+7 **MARK** MOV WORD PTR [COUNT], 0 ↓

Comments

Line number	Comment
1 - 14	Follow the first 14 numbered comments in "Exercise 1," earlier in this appendix. Then press **RETURN** and proceed with line 15 below.
15	To move on to the Process procedure, type **Process** (press **MARK**) PUSH BP saves the value of the base pointer (BP). Collectively, this instruction and the instructions on lines 16 and 17 are the stack prologue. (For details on the format of the stack, see Appendix C, "Stack Format.") Press **DOWN ARROW**
16	MOV BP, SP Press **DOWN ARROW**
17	SUB BP,0 Line 17 subtracts 0 bytes from the stack pointer for local variables. Since Process has no local variables, no space needs to be allocated. Press **DOWN ARROW**
18	Line 18 initializes the count variable.

Script

Line number	Script
19	PROCESS+0D **MARK** MOV DI, 0F12A ↓
20	PROCESS+10 **MARK** MOV SI, 0FD23 ↓
21	PROCESS+13 **MARK** PUSH DS ↓
22	PROCESS+14 **MARK** POP ES ↓
23	PROCESS+15 **MARK** CLD ↓
24	PROCESS+16 **MARK** MOVSW ↓
25	PROCESS+17 **MARK** MOVSW ↓
26	PROCESS+18 **MARK** MOVSW ↓
27	PROCESS+19 **MARK** MOVSW **RETURN**
28	***Process+15 MARK** CLD **.^b** **GO**

Comments

Line number	Comment
19 - 27	To advance through the Process procedure instructions on lines 19 through 27, press **DOWN ARROW** This common instruction sequence indicates a string move: the string 'Finished' is being assigned to the string variable Done. (See the program listing shown in Figure 3–1 in Section 3, "Debugging Session.") The assignment statement done := 'Finished' moves the string 'Finished' one word at a time from the source located at DS:SI (source index) to the destination at ES:DI (destination index), as shown below. **Destination** **Source** ES:DI DS:SI Done := 'Finished'
28	To execute the code up to the point that the MOVSW instructions begin type **process+15** (press **MARK**) The Debugger displays the CLD instruction previously shown on line 23. To set a breakpoint at this location, type a period (.); then press **CODE-B** To execute up to the breakpoint, press **GO**

Script

Line number	Script
29	Exiting Debugger
30	Break at PROCESS+15 in process 0D
31	**8, DS:SI ^d**
32	0DBF2:0FD23 46 69 6E 69 73 68 65 64 Finished
33	**8, ES:DI ^d**
34	0DBF2:0F12A 00 00 00 00 00 00 00 00
35	**cs:ip MARK** CLD **^x**
	#
36	PROCESS+16 **MARK** MOVSW **^x**

Comments

Line number	Comment
29	Exiting Debugger
30	Break at PROCESS+15 in process 0D
31	To look at 8 bytes of memory at the source location, type **8, DS:SI** (press **CODE-D**)
32	The Debugger displays `0DBF2:0FD23  46 69 6E 69 73 68 65 64  Finished`
33	To look at 8 bytes of memory at the destination, type **8, ES:DI** (press **CODE-D**)
34	The Debugger displays `0DBF2:0F12A  00 00 00 00 00 00 00 00` This display shows that no bytes have been moved here yet.
35	To see the next instruction to be executed, type **cs:ip** (press **MARK**) The Debugger displays the CLD instruction. To execute this single instruction, press **CODE-X**
36	Execute the first MOVSW (move word) instruction. Press **CODE-X**

Script

Line number	Script
	#
37	`PROCESS+17` **`MARK`** `MOVSW` **`RETURN`**
38	**`DI`** `→ F12C` **`RETURN`**
39	**`8, es:0f12a ^d`**
	`0DBF2:0F12A  46 69 00 00 00 00 00 00   Fi`
40	**`8, done ^d`**
	`0DBF2:0F12A  46 69 00 00 00 00 00 00   Fi`

Comments

Line number **Comment**

37 Press **RETURN** after the second MOVSW (move word) instruction is displayed.

38 To look at the contents of the DI register,

type **DI** (press **RIGHT ARROW**)

The Debugger displays F12C. (Note that DI now is 2 greater than DI on line 34, indicating that the index pointer has advanced 2 bytes.)

Press **RETURN**

39 To look at 8 bytes of memory starting at the original destination (the value of DI on line 34),

type **8, es:0f12a** (press **CODE-D**)

The Debugger displays

```
ODBF1:0F12A  46 69 00 00 00 00 00 00   Fi
```

The first two characters in the string 'Finished' were moved to the destination as a result of your executing the MOVSW instruction.

40 Because the destination is the address of the public variable done, you could look at the same 8 bytes by typing

8, done (press **CODE-D**)

Try this.

The Debugger display is the same as shown on line 39.

Script

Line number	Script	
41	^c	GO

Comments

Line number	Comment
41	Before exiting the Debugger, clear the breakpoint you set.

Press **CODE-C**

To exit,

press **GO**

The DisplayFile program executes to completion, displaying the *DisplayFile.pas* file to the screen. The string 'Finished' appears following the last file character.

Using the fDevelopement Flag

By setting the fDevelopement flag, you can conveniently detect the error source in any of your program procedures that do not require customized error exit routines. fDevelopement is a public symbol in the CheckErc routine of CTOS.lib. (Note the e in the spelling of fDevelopement.) Use of this flag is limited to the CTOS operations that pass the returned error code to the CheckErc procedure.

To use the fDevelopement flag, declare CheckErc as an external procedure in your program, as shown below:

```
PROCEDURE CheckErc (ercCode: Word); EXTERN;
```

Declare the CTOS operations that return error codes as external functions, for example

```
FUNCTION OpenByteStream (
pBSWA           :  Pointer;
pbFileSpec      :  Pointer;
cbFileSpec      :  Word;
pbPassWord      :  Pointer;
cbPassWord      :  Word;
mode            :  Word;
pBufferArea     :  Pointer;
sBufferArea     :  Word) : ErcType; EXTERN;
```

Then, write the statements calling CTOS operations such that the error code is passed to CheckErc, for example

```
CheckErc (OpenByteStream (ADS BSWA, ADS
'DisplayFile.pas', 14, ADS NULL, 0, mt, ADS buffer,
1024));
```

Under normal circumstances when a nonzero error code is returned, CheckErc calls the FatalError operation. FatalError, in turn, calls ErrorExit, which terminates the program and passes the error code to the exit run file (typically the Executive). (For details on CheckErc, FatalError, and ErrorExit, see the *CTOS Procedural Interface Reference Manual.*)

You can, however, have your program enter the Debugger automatically if you set the fDevelopement flag.

You can set this flag in either of two ways: dynamically, in the memory copy of your program in the Debugger; or statically, by using the **Debug File** command. (For details on Debug File, see Section 11, "The Executive Command: Debug File.")

The following exercise uses the DisplayFile program described earlier in this appendix and in Section 3, "Debugging Session," to demonstrate how to set this flag dynamically. (For details on the files you need, see exercise 1 earlier in this appendix.) With the exception of ReadByte, which has its own error checking routine, all the CTOS operations in the DisplayFile program pass error codes to CheckErc.

The DisplayFile program contains an error. The error may be obvious to you by now, as you were informed of where it is and how to patch it in earlier examples in this appendix. This time, however, you will locate the error yourself by using the fDevelopement flag.

Run the DisplayFile program, and enter the Debugger by pressing **CODE-GO**. Then, follow the script and comments on the next few pages.

Script

Line number	Script
1	**'DisplayFile.sym'** ^f
2	***DisplayFile** ^b GO
	*
3	Exiting Debugger
4	Break at ENTGQQ in process 0D
5	***fDevelopement** ← 00 **0ff** ↓
6	fDevelopement+1 GO
7	Exiting Debugger
8	Debugger called at FATALERROR+0E in process 0D

Comments

Line number	Comment
1	Load the symbol file. Type **'DisplayFile.sym'** (press **CODE-F**)
2	Then, set a breakpoint at the beginning of the program. Type **DisplayFile** (press **CODE-B**) Execute up to the breakpoint by pressing **GO**.
3	Exiting Debugger
4	Break at ENTGQQ in process 0D
5	To display the value of fDevelopement, type **fDevelopement** (press **LEFT ARROW**) The Debugger displays 00, (FALSE) the default setting, which causes a program to terminate and the exit run file to be loaded. To set the flag to TRUE, type **0ff** Then to close the location, press **DOWN ARROW** (or **RETURN**)
6	Press **GO** to execute the program.
7	Exiting Debugger
8	The Debugger exits and is automatically entered again at the location FATALERROR+0E in the FatalError procedure.

Script

Line number	Script
9	**^t**
	0 F088 FATALERROR+0E (286, 0EBF6, 0CB)
	1 F094 ENTGQQ+35
	2 F09A BEGXQQ+98
10	**0CB=**0CB
11	***^r=**203.
12	***^r=**0CB
13	**ENTGQQ+35 MARK** CALL PROCESS ↑

Comments

Line number **Comment**

9 To trace the stack,

press **CODE-T**

The Debugger displays

```
0  F088  FATALERROR+0E (286, 0EBF6, 0CB)
1  F094  ENTGQQ+35
2  F09A  BEGXQQ+98
```

The rightmost value on the top line of this display (0CB) is the error code passed to FatalError. (For details on stack format, see Appendix C, "Stack Format." There is also a brief discussion of the stack supported by an example in the tutorial in Section 3, "Debugging Session.")

10 To change the radix of the error code from hexadecimal (current radix) to decimal, first

type **0CB=**

The Debugger redisplays 0CB.

11 Then to change the radix to decimal,

press **CODE-R**

The Debugger displays the error code in decimal notation. Error code 203 means "No such file."

12 Change the radix back to hexadecimal as shown.

13 The error occurred in the first CTOS operation that returned an error code prior to the call to FatalError. To view the instruction preceding the one at FATALERROR+0E,

type **ENTGQQ+35** (press **MARK**)

The Debugger displays `Call PROCESS`.

To view the previous instruction,

press **UP ARROW**

Script

Line number	Script
14	ENTGQQ+30 **MARK** CALL CHECKERC ↑
15	ENTGQQ+2F **MARK** PUSH AX ↑
16	ENTGQQ+2A **MARK** CALL OPENBYTESTREAM ↑
17	ENTGQQ+29 **MARK** PUSH AX ↑
18	ENTGQQ+26 **MARK** MOV AX,400 ↑
19	ENTGQQ+25 **MARK** PUSH AX ↑
20	ENTGQQ+24 **MARK** PUSH DS ↑
21	ENTGQQ+21 **MARK** MOV AX,0F136 ↑
22	ENTGQQ+20 **MARK** PUSH AX ↑
23	ENTGQQ+1D **MARK** MOV AX,6D74 ↑
24	ENTGQQ+1C **MARK** PUSH AX ↑
25	ENTGQQ+1A **MARK** XOR AX,AX ↑
26	ENTGQQ+19 **MARK** PUSH AX ↑
27	ENTGQQ+18 **MARK** PUSH DS ↑
28	ENTGQQ+15 **MARK** MOV AX, 0FD3C ↑
29	ENTGQQ+14 **MARK** PUSH AX ↑
30	ENTGQQ+11 **MARK** MOV AX,0E ↑
31	ENTGQQ+10 **MARK** PUSH AX ↑
32	ENTGQQ+0F **MARK** PUSH DS ↑
33	ENTGQQ+0C **MARK** MOV AX, 0FD2D ↑
34	ENTGQQ+0B **MARK** PUSH AX ↑

Comments

Line number	Comment
14	The Debugger displays Call CHECKERC. The call to CheckErc indicates that an earlier instruction passed an error code to CheckErc. To view the previous location, press **UP ARROW**
15	The Debugger displays PUSH AX. To view the location preceding PUSH AX, press **UP ARROW**
16	The Debugger displays CALL OPENBYTESTREAM. OpenByteStream is the CTOS operation in question.
17 - 39	To view lines 17 through 39, continue pressing **UP ARROW** You should recognize these instructions from the tutorial in Section 3, "Debugging Session." These instructions push the parameters to OpenByteStream onto the stack. Lines 37 through 39 show the stack prologue, signaling the beginning of the stack frame. (For help in interpreting these instructions, see the tutorial in Section 3. Also see "Looking at the Stack.") There is an error related to a file specification somewhere in the parameters pushed onto the stack. If you examine the parameters to OpenByteStream, you will note that the second and third parameter are the address and count of bytes (pb/cb pair) of a file specification. On line 33, the offset of the file specification (0FD2D) is moved into AX. To view the file specification, you need to examine the first 14 bytes starting at this offset in the data segment.

Script

Line number	Script
35	ENTGQQ+0A **MARK** PUSH DS ↑
36	ENTGQQ+07 **MARK** MOV AX, 0F0A8 ↑
37	ENTGQQ+03 **MARK** SUB SP,4 ↑
38	ENTGQQ+01 **MARK** MOV BP,SP ↑
39	ENTGQQ **MARK** PUSH BP **RETURN**
40	0E, DS:0fd2d **^d** `0DBF2:0FD2D  44 69 73 70 6C 61 79 46 69 6C 65 2E 70 61 DisplayFile.pa`
41	Exiting Debugger

Comments

Line number	Comment
39	Press **RETURN** after displaying the instruction on line 39.
40	To look at the 14 bytes of the file specification, type **0E, DS:0fd2d** (press **CODE-D**) The Debugger displays `0DBF2:0FD2D 44 69 73 70 6C 61 79 46 69 6C 65 2E 70 61 DisplayFile.pa` The complete specification, however, is *DisplayFile.pas* (not *DisplayFile.pa*). The source of the error is the instruction MOV AX, 0E. The instruction only moves 14 words instead of 15. This error cannot be corrected at this point in the Debugger, because the correct file specification is required for successful program execution. It would be easier to make such a correction to your source code; then recompile and relink the program.
41	To exit the Debugger, press **ACTION-FINISH** **ACTION-FINISH** terminates the program and loads the exit run file.

When You Get a Nonfatal GP Fault

On a protected mode operating system, your protected mode program is prevented from accessing memory outside of the segments defined by the descriptor tables. If, for example, your program attempts to use an invalid selector or to exceed a segment limit, a nonfatal general protection (GP) fault is generated and the system passes control to the Debugger in simple mode. For your convenience in debugging, you can configure your system so that the Debugger is entered automatically when a GP fault occurs. See Appendix F, "Configuration Options for the Debugger," for details.

When a fault is generated with your system configured this way, you enter the Debugger automatically. A three-line message of the following form is displayed (the first two lines):

> Debugger XX (simple mode)
> GP fault at *addr* in process X

The first line of the message indicates your Debugger version (XX) and the Debugger mode (for example, simple mode). The second line shows the cs:ip (*addr*) and the process identification number (X) of the process that caused the fault. The third line is a message that may help you understand why the GP fault may have occurred, as shown in the examples of messages below:

```
Invalid selector:  0004
Null selector or limit violation
```

To make debugging easier, load the symbol file for your program.

> Type **'FileName.sym'** (press **CODE-F**)

For details on symbol files, see "Symbol Files" in Section 4, "General Purpose Debugger Functions and Features."

Once you have loaded symbols, you can approach isolating the GP fault in your program. Following are suggestions to help you with this process.

Invalid Selector

If the Debugger displayed a message, such as

```
Invalid selector:  0004
```

the invalid selector is identified for you. What you need to find out is which address contains this selector and where the faulty address is used in your program.

First, however, if you have never used the **CODE-V** command (described in Section 7, "Display Commands"), this is an opportunity to try out the command to see what a descriptor looks like for an invalid selector. Type the selector number indicated in the Debugger message and press **CODE-V**, for example

0400 (press **CODE-V**)

The Debugger displays the segment descriptor, such as

```
iSn   sn   base limit ar p
0060 0400 0FC1F0 02F2 9B 0 invalid type
```

The segment description is 'invalid type,' which indicates that there is something wrong with the selector.

If, on the other hand, you used **CODE-V** with a valid selector, the Debugger would display a description, such as

```
iSn   sn   base  limit ar  p
0060 0400 0FC1F0 02F2 9B 0 code, non-conforming, readable
```

For details on all the fields displayed by **CODE-V** command, see the description of CODE-V in Section 7, "Display Commands."

Determining Which Address Caused the Fault

To locate the faulty address, display the instruction at cs:ip,

type **cs:ip** (press **MARK**)

The Debugger displays the instruction in which the fault occurred. Typically this instruction is

```
LES BX,WORD PTR [addr]
```

where [*addr*] is the location of the faulty address. This instruction loads the ES and BX registers with the segment address and offset, respectively, of (what is assumed to be) an address located at [*addr*].

Examples of possible faulty addresses are:

pMessage	Is a global variable you declared public in the static data segment (at the beginning of your program). A public variable immediately tells you which address contains the invalid selector.
BP-0A	Is the location on the stack of a local variable. "Exercise 2," earlier in this appendix, describes how you can identify stack variables by their relative locations to the base pointer (BP). For details, see the comments accompanying steps 18 through 22 in exercise 2.
41F5	Is a global variable that you did not declare public in the static data segment.

Locating the Fault in Your Program

There are several ways you can approach locating where the faulty address was used in your program. Following are some suggestions.

Once you have displayed the instruction as cs:ip, as described earlier, you can use the **UP ARROW** or **DOWN ARROW** keys to hone in on the instructions leading up to or immediately following the instruction. These instructions should provide you with some clue as to where the address variable is being used in your program.

If the fault occurred in a public procedure, the locations of the instructions you view as you use the **UP ARROW** or **DOWN ARROW** keys can provide you with the approximate location of where the fault occurred. If, for example, you pressed **DOWN ARROW** and the Debugger displayed the location

ProcessName+3

you would know that the fault occurred at the beginning of the public procedure ProcessName. If, on the other hand, the location displayed was

ProcessName+29

you would know that the fault occurred somewhere within the ProcessName procedure but not at the beginning.

A public variable immediately tells you the symbolic name of the faulty address. If the variable is on the stack, you can look at the stack. For details on interpreting the stack, see "Looking at the Stack." What you need to find is some clue on the stack (such as a word value passed) to determine where the local variable is being used in your program.

Faults caused by variables that are not declared public are more difficult to detect. If absolutely necessary, you can declare these variables publicly for easier identification.

In all cases, examine your source code to check the variable within the context of how it is used.

Null Selector or Limit Violation

If on a GP fault, the Debugger displays the message

```
Null selector or limit violation
```

you need to determine which of these is the reason the fault occurred.

Using a Null Selector

A null selector is a selector with the value 0. Your program can load a null selector into a segment register without generating a fault. However, when your program attempts to use the address created with the selector, it GP faults because the selector is invalid.

Suppose, for example, you declare a pointer type of variable at the beginning of your Pascal program in the static data, such as

```
VAR
.
.
.
pCount:    Pointer;
```

Then, if you use the variable, such as in a function call to WriteBSRecord, as shown below

```
erc := WriteBSRecord(pBSWA, ADS data, 16, pCount);
```

your program would GP fault in WriteBSRecord, because pCount was passed with either a null selector or an uninitialized pointer.

To correct this fault, you can declare a variable in your program's static data, such as

```
VAR
.
.
.
junk: WORD;
```

Then, assign pCount to this address before you use pCount in WriteBSRecord, for example

```
pCount:= ADS junk;

erc := WriteBSRecord(pBSWA, ADS data, 16, pCount);
```

Otherwise, you can use a procedure such as AllocAreaSL to allocate and initialize a segment for pCount, for example

AllocAreaSL(16, ADS pCount);

To determine if your program faulted because of a null selector, examine the contents of the ES and DS registers.

Type **ES** (press **RIGHT ARROW**)

Type **DS** (press **RIGHT ARROW**)

On 386 processors, it is a good idea to examine the contents of the FS and GS registers.

If the Debugger displays the null value 0 for either of these registers, you can suspect that the register was loaded with an uninitialized pointer.

Exceeding a Segment Limit

Limit violations typically occur when your program attempts to move data to or from a location that is outside of the intended segment.

To start your investigation of a segment limit violation, display the instruction at cs:ip.

Type **cs:ip** (press **MARK**)

Typically, the Debugger displays a type of string move instruction. For example, if your source program is written in Pascal, you may see an instruction, such as

PROCESS+16 **MARK** MOVSW

In your investigation, you need to be concerned with the contents of following registers:

ES
DS
SI
DI
CX

DS:SI and ES:DI contain the addresses of the source and destination, respectively, of a type of string move instruction. If you have not already done so, you should try the hands-on example in "String Moves," earlier in this appendix. The example demonstrates how the index register pointers SI and DI advance as you execute MOVSW (move byte) instructions in the Debugger to move a word at a time from the source to the destination.

The CX register contains the count of bytes to be moved. To examine the contents of CX,

type **CX** (press **RIGHT ARROW**)

If the value displayed appears unusual (for example, if it is a value such as FFFF), you can suspect a problem with exceeding the segment limit.

Use **RIGHT ARROW** to display the contents of ES and DS.

Type **ES** (press **RIGHT ARROW**)

Type **DS** (press **RIGHT ARROW**)

You can use the value of ES or DS (such as, 039B) with the **CODE-V** command to examine the segment limit, for example

type **039C** (press **CODE-V**)

The Debugger displays the descriptor for 039C, such as

```
iSn   sn   base   limit ar  p
0060 039C 0FC1F0 02F2 9B 0 code,non-conforming,readable
```

Then, you can compare the limit value with that contained in SI or DI.

Say, for example that you used the AllocMemorySL operation to allocate a segment of memory for an array. Then, as your program executed instructions to move the elements into the array, it GP faulted. This fault might occur if your program attempted to move more elements into the array than there was space allocated. If you then compared the value of the destination index DI to the segment limit displayed by **CODE-V**, you would find that DI contained a value that exceeded the limit.

This fault also might occur if more bytes were moved from the source than were actually there. In this case, SI would contain a value greater than the limit displayed by **CODE-V**.

Appendix E
Debugger Swapping

Debugger Swapping

The Debugger requires approximately 80K bytes of memory (check the Release Notice for your system to ascertain the correct number). However, under some circumstances (for example, if you are debugging the Word Processor), you can debug a program that occupies all of memory, theoretically leaving no room for the Debugger.

The Debugger manages memory according to these procedures:

- If enough memory is available, the Debugger uses memory that is not used by the other processes.
- If enough memory is not available, the Debugger swaps out part of the user's program, provided swapping out is possible.

When the Debugger swaps out part of the user's program, program execution is not affected.

Swapping can occur only when the Debugger is in simple mode. In simple mode, all user processes are suspended when they reach a breakpoint. (See also Section 9, "Debugger Modes.")

Note: *The Debugger swapping mechanism described in this appendix is supported on real mode CTOS operating systems only. This swapping is not related or connected in any way to the Virtual Code Management facility, which is sometimes called the Swapper in other documentation.*

Appendix F
Configuration Options for the Debugger

What are the Debugger Configuration Options?

There are several optional ways you can configure your software for using the Debugger. All of the configuration options described in this appendix are contained in the protected mode system configuration file (the default for workstations is *[Sys]<Sys>Config.sys*; the default for shared resource processors is *[Sys]<Sys>SrpConfig.sys*). For additional information on the system configuration file options, see the *CTOS System Administration Guide.* See Appendix B, "Shared Resource Processor Debugging," in this guide for more information on the shared resource processor files.

One option allows you to configure the real mode Debugger for use under the Context Manager.

Using an Extended Crash Dump File

An extended crash dump is a dump of the extended memory on a protected mode operating system. The way that the extended crash dump process works is described in Appendix G, "Extended Crash Dump Process." It is recommended you read that appendix first so that you understand what an extended crash dump is, why you would need an extended crash dump file, as well as how you would go about sizing the file correctly. Setting up crash dumps is also described in the *CTOS System Administration Guide.*

In addition, you should read Section 11, "The Executive Command: Debug File." That section describes how you can approach debugging a crash dump file using the **Debug File** command. However, to use **Debug File**, you need to have created a crash dump file first, so you can obtain a byte-for-byte disk file copy of the System Image at the time of a crash. Alternately, you have an opportunity to create this file at reboot after a system crash. (For details on extended crash dump files, see the description of the **Extended Crash Dump** command in the *CTOS Executive Reference Manual.*)

Workstation Crash Dump File Options

On protected mode workstations, you can capture crash dumps to a file using the following configuration options in the system configuration file. You can use the options to conserve disk space and still have an extended crash dump file. These options are

```
:SuppressAutoDump:
:ExtCrashDumpFile:
:CrashDumpFile:
:ExtCrashVDMFile:
```

Ways you can use each of these options in a disk saving way are described below. See the *CTOS System Administration Guide* for a description of all the *Config.sys* options.

:SuppressAutoDump:	Directs the operating system to suppress the dumping of extended memory.

Use. To use this option (default is No), edit *Config.sys* as follows:

```
:SuppressAutoDump: Yes
```

Recommendation. This option is useful when you require an extended crash dump file but, because you are conserving disk space, the file does not exist.

You can create an extended crash dump file using the **Create File** command. Then execute an extended dump using the **Extended Crash Dump** command. Following the extended dump, rebootstrap your system to regain the use of extended memory. When you are done using the extended crash dump file, you then can delete it.

Note: *Do not use this option unless you are using :ExtCrashVDMFile:.*

:ExtCrashDumpFile:	Allows **Extended Crash Dump** to find the extended crash dump file when it is not created in the default directory [Sys]<Sys>.

Use. As an example of using this option, you could edit *Config.sys* as follows:

```
:ExtCrashDumpFile: [d2]<dumps>ExtCrashDump.sys
```

This entry directs Extended Crash Dump to use *[d2]<dumps>-ExtCrashDump.sys* rather than *[Sys]<Sys>ExtCrashDump.sys.*

Recommendation. This option is useful for saving disk space on the system volume by locating the extended crash dump file on another volume.

:CrashDumpFile:	Allows **Extended Crash Dump** to find the crash dump file when it is not in the default directory [Sys]<Sys>.

Use. As an example of using this option, you could edit *Config.sys* as follows:

```
:CrashDumpFile: [d0]<Sys>CrashDump.sys
```

This entry directs **Extended Crash Dump** to use the file *[d0]<Sys>CrashDump.sys* rather than the file *[Sys]<Sys>CrashDump.sys.*

Recommendation. This option is useful if you require a crash dump file to be located in a different disk. As examples, you can use this option in either of the following situations: if the system volume is [d1] but a nonzero length *CrashDump.sys* file exists on [d0], or if the system volume is at the server but a valid *CrashDump.sys* file exists on a local hard disk.

:ExtCrashVDMFile:	Specifies the video display manager to run while doing an extended crash dump.

Use. To use this option, edit *Config.sys* as follows:

```
:ExtCrashVDMFile: [Sys]<Sys>Vdm_Ch.run
```

Recommendation. By default, the operating system installs a dummy video display manager before doing an extended crash dump. Use this option in conjunction with the :SuppressAutoDump: option.

Shared Resource Processor Crash Dump File Options

On shared resource processors, the *SrpConfig.sys* file includes two options:

:Boot:
:CrashDumpPath:

:Boot:	Determines whether or not a crash dump is generated.

Use. This option has three parameters, shown below, and is discussed in detail in the *CTOS System Administration Guide*. The Dump parameter determines whether or not a crash dump is generated and is described here.

:Boot: (Processor = *XPnn*, OS = *Filespec*, Dump = Yes, No, or Lowmem)

where the values in the Dump parameter are as follows:

No	This means that no crash dump is generated for that board. This is the default.
Yes	This means the system should automatically dump all of the memory for the board to the corresponding crash dump file.
LowMem	This is the same as specifying Yes, except in the case of a board that has more that 4M bytes of memory. For example, if a GP board has 64M bytes of memory, the crash dump may not fit on the disk where the crash dumps are being created. By using LowMem, you can run the Extended Crash Dump utility and specify a path for the crash dump file where there is enough free disk space. You can also use LowMem if Yes fails to create the crash dump file.

:CrashDumpPath:	Allows you to determine where the crash dump will be created. The volume specified must be a device that is attached to the server processor. By default, all crash dumps are created in [Sys]<Sys>.

Use. To use this option, edit SrpConfig.sys as follows:

:CrashDumpPath: [*volume*]<*directory*>

Using the Debugger on a Shared Resource Processor

Because a shared resource processor has multiple processor boards, the way you configure the Debugger for a shared resource processor is different from the way you configure it for a workstation.

If you want to debug using Basic ATE on a shared resource processor, first specify to load the Debugger with the :LoadDebugger: option (see "Suppressing the Debugger"). Then add the following option to the processor section of the configuration file:

:DebugPort:

This entry defines the RS-232-C line to be used by the Debugger :DebugPort: option. See Appendix B, "Shared Resource Processor Debugging" for more information.

Suppressing the Debugger

The Debugger consists of a resident and a nonresident portion. The resident portion is located in the operating system unless your system is configured to exclude it. (See your release documentation for details.) The nonresident portion of the real mode Debugger is swapped into memory as needed. With the protected mode Debugger, however, the nonresident portion is always loaded into memory unless you specify that loading be suppressed with the configuration file option :SuppressDebugger: (default is No).

:SuppressDebugger:	Suppresses loading of the Debugger.

Use. To use this option, edit *Config.sys* as follows:

```
:SuppressDebugger: Yes
```

This option applies to workstations only.

On shared resource processors, the Debugger is automatically suppressed unless you specify to load it into memory with the configuration file option :LoadDebugger: (default is No):

:LoadDebugger:	Specifies whether or not to load the Debugger.

Use. To use this option, use the default of No as follows to suppress loading the Debugger:

```
:LoadDebugger: No
```

See Appendix B, "Shared Resource Processor Debugging" for more information.

Entering the Debugger on a GP Fault

On a protected mode operating system, your protected mode program is prevented from overwriting the memory of other programs. If, for example, your program attempts to use an invalid selector or to exceed a segment limit, your program GP faults. As a convenience to you while you are debugging, you can configure your system so that the Debugger is entered automatically on a GP fault with the system configuration file option :EnterDebuggerOnFault: (default is No).

:EnterDebuggerOnFault:	Provides automatic entry into the Debugger when your program generates a GP fault.

Use. To use this option, edit the system configuration file as follows:

```
:EnterDebuggerOnFault: Yes
```

Note: *Because there is no video on a shared resource processor unless Administrator ClusterView is running, this should be used only when debugging a known GP fault.*

Recommendation. This option is recommended if you are programming on a protected mode system. For details on how to locate the cause of the fault, see "When You Get a Nonfatal GP Fault," in Appendix D, "Debugger Tips." If you are not using Administrator ClusterView, you have no way of knowing about the fault.

On a shared resource processor, if this option is disabled and there is a fatal GP fault, the system continues in a manner which depends on three conditions: which board crashes, the front panel keyswitch position, and the WatchDogState. If the server processor crashes and the keyswitch is in the Normal or Remote position, the system reboots. If the keyswitch is in the Manual position, the crash error code is displayed on the LEDs, and 50 is displayed on the front panel.

If a nonserver processor board crashes, it displays the crash error code on its LEDs. In addition, the WatchDog feature of the server processor is invoked and one of three things will happen:

- If the value of WatchDogState is None, the server processor allows the system to continue as best it can.
- If the value of WatchDogState is SetFlag, the server processor displays 40 on the front panel to indicate that a board has crashed; it allows the system to continue.
- If the value of WatchDogState is Crash, the server processor crashes with error code 107. The system reboots if the keyswitch is in the Normal or Remote position.

For more information about the WatchDogState feature of the server processor, see the *CTOS System Administration Guide*.

Using the Debugger Under the Context Manager

Note: *This subsection applies only to workstations running CTOS I.*

As stated earlier, the nonresident portion of the real mode Debugger is swapped into memory as needed. Swapping occurs under the Context Manager by default unless you configure the :DebuggerSize: option (default is 0) in your Context Manager configuration file *CMConfig.sys*, so that enough memory is reserved for the Debugger to remain in memory at all times. This option applies to workstations only.

:DebuggerSize:	Allows you to specify the amount of memory (in K bytes) for using the Debugger under the Context Manager.

Use. To ensure that there will always be enough memory for the Debugger (so swapping does not occur), configure the :Debuggersize: entry in *CMConfig.sys* as follows:

```
:DebuggerSize: 75
```

Recommendation. Using CODE-I breakpoints under the Context Manager will not work if the real mode Debugger is allowed to swap. Therefore, you must specify 75K bytes for the Debugger size. It is further recommended that you specify this size if you want to use **CODE-X** to single step instructions efficiently. Otherwise, the Debugger may swap out of memory each time you execute an instruction.

Using a Second Screen for Debugging

You can connect a second monitor to your system and use it for debugging.

Use. Configure the :VGACharMapDebugger: entry in the *Config.sys* file as follows:

```
:VGACharMapDebugger: yes
```

See Appendix J, "Debugging on a Second Monitor," for more information.

Appendix G
Extended Crash Dump Process

What is the Extended Crash Dump Process?

If your system crashes, you can obtain a *crash dump*, or snapshot, of what memory looked like before the crash. Once you have obtained the crash dump as a disk file, you can analyze the source of the crash by using the **Debug File** command described in Section 11, "Executive Command: Debug File."

This appendix describes the extended crash dump process, sizing of crash dump files, and reasons you may need to create a crash dump file. For details on ways to conserve disk space, see Appendix F, "Configuration Options For the Debugger." Crash dumps for real mode operating systems are described in the *CTOS System Administration Guide*.

Different hardware platforms have different dumping capabilities. Some platforms are capable of dumping only the first megabyte of RAM. Other platforms are capable of dumping all existing RAM. Still others have no crash dump capabilities. In the last case, the operating system performs the dump.

Table G–1. Hardware Capabilities Table

Hardware Platform	Dumps 1Mb RAM	Dumps all RAM	Dumps no RAM
SG–5000	X	X	
SG–2000	X	X	
GP (SRP)	X	X	
B39	X		
B38Exp	X		
EISA/ISA			X
All Others	X		

Systems Capable of Dumping the First Megabyte

On systems that dump only the first megabyte, the operating system reboots after the dump but restricts itself to the first megabyte of memory, ensuring that the (as yet) undumped memory remains unchanged. In this mode, the operating system shrinks its memory usage (for example, the debugger is not loaded, and certain data structures are reduced), and it loads only the Video and the Extended Crash Dump program. The Extended Crash Dump program combines the crash dump file created by the bootstrap ROM (the first megabyte of RAM) and the memory above one megabyte and creates a new file. This new file is the extended crash dump file, which is an entire memory image. The bootstrap ROM always dumps to the first disk (from the left) which contains a nonzero *CrashDump.sys* file. The Extended Crash Dump program determines where the original crash dump file is and which file is to be the extended crash dump file by reading the system configuration file. The entries are

```
:ExtCrashDumpFile: [d1]<Sys>ExtCrashDump.sys
:CrashDumpFile: [d0]<Sys>ExtCrashDump.sys
```

Systems Capable of Dumping all Memory

On systems capable of dumping all of memory, the system can be configured one of two ways:

- So that the bootstrap ROM performs the entire crash dump
- So that the extended crash dump process described above takes place

To do this, set the size of the *CrashDump.sys* file. If the size of the *CrashDump.sys* file is equal to or greater than the size of existing RAM, the bootstrap ROM creates a complete crash dump and no extended crash dump process is necessary. If the *CrashDump.sys* file is smaller than existing RAM, the bootstrap ROM dumps RAM equal to the size of the crash dump file, and the operating system dumps the rest. The minimum useful size for the crash dump file is one megabyte.

Systems Not Capable of Performing Crash Dumps

On systems that are not capable of dumping at all (EISA/ISA), the operating system assumes the task. At crash time, the operating system copies the first megabyte to the crash dump file, reboots, and the extended crash dump process begins.

Sizing Crash Dump Files

For an extended crash dump to succeed on a workstation or shared resource processor, the crash dump file must be large enough to hold the memory that the bootstrap ROM dumps. There should be 2048 sectors in your crash dump file for each megabyte of your memory configuration. On workstations, *CrashDump.sys* must be at least 2048 sectors so that the bootstrap ROM can dump the first 1M bytes of memory.

On shared resource processors, the crash dump file must be at least 1536 sectors on a real mode board and 8192 sectors on a protected mode shared resource processor board.

In addition, for workstations, if *CrashDump.sys* is not large enough to hold the entire memory image, *ExtCrashDump.sys* must exist and be large enough to hold the entire memory image.

When You May Require an Extended Crash Dump File

An extended crash dump file is provided to assist you in the following situations:

- You may not be able to create a crash dump file larger than 4096 sectors. This situation can happen, for example, if during initialization the number of bad spots detected results in an insufficient number of contiguous sectors. If more that 2M bytes of memory are present, the use of the extended crash dump file is recommended.
- Because the crash dump file is a dedicated file created by the Executive **Format Disk** command at initialization, it cannot be deleted or resized without reinitializing the entire disk. To conserve disk space, therefore, you may find that you need to configure the crash dump file to be of reasonable size. In those cases where you need more space than the crash dump file allows, you may create a larger extended crash dump file, use it, and then delete it.

How the Extended Crash Dump Process Works

In the paragraphs that follow, the extended crash dump process is described for workstations and for shared resource processors.

Workstations

An extended crash dump is a dump of the memory beyond 992K bytes on a protected mode system.

In the discussion that follows, the default directory for the crash dump files is [Sys]<Sys>.

In addition, if a local hard disk is not available and the workstation is attached to a cluster, the crash dump files have the prefix ws>.

Following a system crash, the bootstrap ROM dumps the first 992K bytes of memory to the file *CrashDump.sys*, which must be created when the disk is formatted with the **Format Disk** command. (For details on Format Disk, see the *CTOS Executive Reference Manual*).

After the dump, the bootstrap ROM reloads the operating system. The system initializes using only memory within the first 992K bytes. Following installation of a dummy Video Display Manager, the operating system invokes the Extended Crash Dump utility (hereafter called Extended Crash Dump), to dump memory beyond 992K bytes. At completion of the extended dump, Extended Crash Dump reenters the bootstrap ROM. The bootstrap ROM then reloads the operating system, which initializes using all available memory.

A dummy video display manager is a version that does not display characters on the screen. This is necessary because of the restricted amount of memory available to programs during the extended crash dump process. If you have a system that supports character-mapped video, you may be able to use a character-mapped version of VDM during extended crash dumping. To do so, use the :ExtCrashVDMFile: option in *Config.sys.*

For an extended crash dump to succeed, your extended crash dump file must be large enough for your memory configuration. (See the description of the **Extended Crash Dump** command in the *CTOS Executive Reference Manual* for details on creating an extended crash dump file. Sizing of crash dump files is discussed in "Sizing Crash Dump Files.")

If Extended Crash Dump fails to locate a crash dump file of the required size, the failure is logged and no extended dump occurs.

Extended Crash Dump first attempts to use the file *CrashDump.sys* as the extended crash dump file. If *CrashDump.sys* is of the required size, Extended Crash Dump simply copies extended memory to *CrashDump.sys.*

If, however, *CrashDump.sys* is too small, Extended Crash Dump attempts to use the file *ExtCrashDump.sys.* If *ExtCrashDump.sys* is of the required size, Extended Crash Dump first copies the contents of *CrashDump.sys* to *ExtCrashDump.sys* and then copies extended memory to *ExtCrashDump.sys.* Otherwise, the extended crash dump fails.

Shared Resource Processors

Following a system crash (crash of an individual processor board) or a fatal general protection (GP) fault, the bootstrap ROM dumps the first 4M bytes of memory to the file *CrashDump.sys*, which must be created when the disk is formatted with the **Format Disk** command. (For details on Format Disk, see the *CTOS Executive Reference Manual.*)

After the dump, the bootstrap ROM reloads the operating system. The system initializes using only memory within the first 4M bytes. Following installation of the Video Display Manager, the operating system invokes Extended Crash Dump to dump memory beyond 4M bytes. At completion of the extended dump, Extended Crash Dump reenters the bootstrap ROM. The bootstrap ROM then reloads the operating system, which initializes using all available memory.

Extended Crash Dump first attempts to use the *CrashDump.sys* as the extended crash dump file. If the crash dump file is of the required size, Extended Crash Dump simply copies extended memory to it.

If, however, *CrashDump.sys* is too small, Extended Crash Dump attempts to use the file *ExtCrashDump.sys*. If *ExtCrashDump.sys* is of the required size, Extended Crash Dump first copies the contents of the crash dump file to the extended file and then copies extended memory to it. If the file is not large enough or does not exist, Extended Crash Dump attempts to create it.

Extended Crash Dump logs the name of the extended crash dump file in the system Log file.

Crash Dump File Naming and Allocation

Names of crash dump files are different for the server processor than they are for the other boards of a shared resource processor. In this section, the other board is called a nonserver processor.

- For a server processor, the name of the crash dump file is *CrashDump.sys*.

  Like the workstation crash dump file, this file is used by the bootstrap ROM and has to be allocated when the disk is formatted.

- For a server processor extended crash dump file (386 only), the file name is *ExtCrashDump.sys*.

  The server processor extended file is created or expanded as needed.

- For a nonserver processor, the crash dump file name is: *[volume]<directory>X*P*nn.crash*

where:

X	is one of: C, D, F, G, S, or T.
nn	is the processor ordinal position within the shared resource processor.
[*volume*]<*directory*>	is defined by the :CrashDumpPath: option in the shared resource processor system configuration file. (See Appendix F, "Configuration Options for the Debugger," for details.)

For example, *[Sys]<Sys>GP00.crash*. The nonserver processor files are created or expanded as needed.

Crash Detection by the Bootstrap ROM

In order for any dumping to occur, the bootstrap ROM must detect upon reset that the board crashed previously. It does this by checking memory for a valid system time and a nonzero system error buffer. If these conditions are met, the bootstrap ROM proceeds in one of two ways. On the server processor, the bootstrap ROM dumps low memory to CrashDump.sys and sets a flag indicating that a dump occurred. On nonserver processor boards, the bootstrap ROM sets the field bBootCommand in the local CPU descriptor table (CDT) so the server processor will know that it can dump the board.

Note: *If the conditions for dumping are not met, the bootstrap ROM will not preserve the state of memory and there will be no dump of that board.*

Crash Dumps, Nonserver Processor. If Dump is set to No, or if there is no Dump parameter, no dumping occurs. (See the description of the Dump parameter values to the :Boot: option in Appendix F, "Configuration Options for the Debugger.") The distinction between Yes and LowMem depends on whether the board is real mode or protected mode.

For a real mode (186) nonserver processor, the Dump values Yes and LowMem have the same meaning; that is, the server processor dumps the memory of the board into the crash dump file.

For example, assume FP01 is being booted, that Dump=Yes, and that the default crash dump path is [Sys]<Sys>. (Note that the path can be changed using the :CrashDumpPath: option.) If FP01 crashes, its memory is dumped to *[Sys]<Sys>FP01.crash*. Because all real mode boards have 768K bytes of memory, the file will be 1536 sectors.

The setting of the Dump parameter for a real mode board has no affect on automatic rebooting.

Dumping of protected mode (386) nonserver processor boards occurs in two stages. The server processor dumps the low 4M bytes to the crash dump file. The board is booted using only its low memory, and it proceeds according to the Dump value specified. This two-stage dumping sometimes requires automatic rebooting.

Note: *Because a board must be booted to access the memory above 4M bytes, a full dump of a board of more than 4M bytes is possible only when that board is booted.*

If Dump is set to Yes, the server processor dumps the first 4M bytes and the board boots in the first 4M bytes. If a board has only 4M bytes of memory, there is no need to do any extended dumping.

If the board has more than 4M bytes of memory, it dumps its extended memory and indicates to the server processor that it needs to be rebooted to regain the use of memory above 4M bytes.

The extended dumping is done into the same file that the server processor used (XP*nn*.crash), with the size being increased as needed. If the file cannot be made large enough, extended dumping does not occur and the error is logged. The board indicates to the server processor that it still needs to be rebooted.

For example, assume GP02 has 16M bytes of memory and that it crashes. Also assume that Dump=Yes and CrashDumpPath is set to [Sys]<Crash>. When the system is rebooted, the server processor creates *[Sys]<Crash>GP02.crash* with a size of 8192 sectors. It then dumps the first 4M bytes of GP02 into this file. When GP02 boots, it changes the size of the file to 32768 sectors and dumps extended memory from 4M bytes to 16M bytes into the file.

If Dump is set to LowMem, the server processor dumps the first 4M bytes of the board's memory to the crash dump file. When the board boots, the operating system comes up using only the first 4M bytes. If there is more than 4M bytes, the rest of memory is left in the state it was in preceding the crash. You then have the option of running the Extended Crash Dump utility. The system must be rebooted manually to allow the board to regain the use of the memory above 4M bytes.

Crash Dumps, Server Processor. The server processor may also have a boot line in the shared resource processor configuration file. The interpretation is different because the bootstrap ROM is responsible for the initial dumping if a crash is detected. The configuration file can control only extended dumping. As in the case of nonserver processor boards, the action taken depends on the type of board.

For a real mode (186) server processor, the bootstrap ROM dumps all of memory (768K) to *CrashDump.sys*. The Dump parameter in the boot line is ignored. If *CrashDump.sys* does not exist or is too small, no dumping occurs.

For a protected mode (386) server processor, if Dump is set to Yes, the bootstrap ROM dumps the first 4M bytes to *CrashDump.sys*, and the board boots in 4M bytes (above 4M bytes preserved). If *CrashDump.sys* does not exist or is too small, no dumping occurs. When the board boots, if there is only 4M bytes of memory, no extended dumping is necessary.

If there is more than 4M bytes and the bootstrap ROM was successful in dumping the first 4M bytes, the operating system proceeds as follows:

- If *CrashDump.sys* is large enough to hold the entire crash dump (for example, 32768 sectors for a 16M byte board), the dump of extended memory is appended to this file.
- In some cases, *CrashDump.sys* may not be large enough, *ExtCrashDump.sys* is created or expanded as needed. *CrashDump.sys* is copied into it and a dump of extended memory is appended to this new file.

After dumping of its memory has been completed, the server processor continues with inititalization, including dumping and booting other boards in the system, and it initiates an automatic reboot when all of the boards are ready.

If Dump is set to LowMem, the bootstrap ROM dumps the first 4M bytes to *CrashDump.sys*. The board comes up in 4M bytes, and you can run the Extended Crash Dump utility. This is similar to the LowMem case of a nonserver processor in protected mode described earlier.

Analyzing Crash Dumps

Analyzing a crash dump is like debugging a "live" system, except that the dynamic tools (setting breakpoints, for example) are not available to you. Inspecting a crash dump is forensic; all the clues of the failure must come from inspecting the frozen state of a dead system.

To start, use the **Debug File** command on the crash dump file. The appropriate prompt (a filled-in square in protected mode, an empty square in real mode) is displayed. If the prompt is not consistent with the system that crashed (for example, if the empty square displays when the system that crashed was running a protected mode operating system), it is likely that the extended crash dump process failed for some reason. See the rest of this section for troubleshooting information.

The first thing to inspect is SysErrorBuf, the operating system's record of the crash. SysErrorBuf is an eight-word descriptor of the crash. For more information, see the *Status Codes Manual.* To look at SysErrorBuf, type

20:48 (press **CODE RIGHT ARROW**)

The first word is the error code of the crash. Press **DOWN ARROW**. The next word is the process or TSS that caused the crash. Next, set the PR register of this process or TSS. Now you are in the context of the process that caused the crash. You can look at its registers, its current instruction (by typing **cs:ip MARK**), its stack, and so on, just as you would during live debugging.

Once you know the process that caused the crash, you can associate the process with a program by using the **CODE-S** command. If the program is something you are familiar with, you can load its symbol file and look at the process state symbolically. Sometimes the program that crashed did so as an indirect result of the action of another program. The **CODE-S** command is useful for inspecting the mix of running programs.

Caveats

To do the extended crash dump process, the operating system, the video, and the Extended Crash Dump program must be coresident in one megabyte. Depending on the operating system and video being used, this may not be possible at all times. If this occurs, you can direct the operating system to load a very small version of video (Vdm_dmy.run). When this version of video is installed, the video hardware is not updated, so you will not see the Extended Crash Dump program run. However, it will now fit in memory and the process will complete (you will see the disk activity light stay on while the dump is being made).

It is also possible to take crash dumps on systems that have no disks. In this case, the server file system is used. The crash dump file becomes *[!sys]<sys>ws>CrashDump.sys* instead of *<sys>CrashDump.sys* on the first local disk.

Appendix H
Debugger Features Matrix

Debugger Features

Some of the Debugger features described in this guide were introduced with different versions of the operating system. Table H–1 is provided to help you determine which features and commands are supported by the Debugger on your system.

Table H–1. Debugger Features Matrix

Feature/Command	CTOS I 3.4	CTOS/XE 3.4	CTOS II 3.4	CTOS III 1.0
Code-B	X	X	X	X
Code-V		X	X	X
Code-S		X	X	X
Context Manager/VM **Action keys**[2]		X	X	X
Wild card symbols		X	X	X
Programmable function keys		X	X	X
Online help file		X	X	X
Code-W		X	X	X
Virtual 8086 support[1]		X	X	X
Code-=[1]		X	X	X
Code-E		X	X	X
Code-Q		X	X	X
Code-U <userNum>		X	X	X
Code-[			X	
Code-Y				X
Code-Z				X
Resource Symbols				X
DeBugOp				X

[1]80386 microprocessor-based operating systems only.

[2]Context Manager/VM, Version 2.0 or higher.

Appendix I
Debugger Application Programming Interface

Introduction

This appendix describes the application programming interface for the Debugger. It supports various application debuggers, including those you want to create for or port to CTOS. It allows your program to do common debugging operations such as reading and writing memory and registers, setting breakpoints, converting between symbolic and binary program addresses, and doing instruction assembly and disassembly.

DebugOp

DebugOp (pbCmd, cbCmd, pbReqBuf, cbReqBuf, pbRespBuf, cbRespBuf): ercType

Description

A single interface allows an application to request debugger services and/or receive notification of debugger events. It contains multiple command options that request activities such as reading and writing memory contents, setting breakpoints, stepping through memory, translating logical addresses to physical addresses, among others.

Procedural Interface

DebugOp (pbCmd, cbCmd, pbReqBuf, cbReqBuf, pbRespBuf, cbRespBuf): ercType

where

pbCmd

is the address of a command descriptor. The format of the command descriptor is:

Offset	Field	Size	Description
0	usernum	2	The user number of the user being debugged. The contents are command specific. See "Command Types," below.
2	pid	2	The process id of the user being debugged. The contents are command specific. See "Command Types," below.
4	ty	2	The type of command, which is a number from 0 to 25. See "Command Types," below.
6	value	2	A command specific word value.
8	ra	4	A command specific quad value, usually the offset of a segmented address when used with sn.
12	sn	2	A command specific word value, usually the selector of a segmented address.

cbCmd

specifies the size of the command descriptor.

pbReqBuf

is the address of information provided to the debugger service (use depends on cmd.ty).

cbReqBuf

specifies the size of ReqBuf.

pbRespBuf

is the address of information provided to the debugger service (use depends on cmd.ty).

cbRespBuf

specifies the size of RespBuf.

Command Types

The **DebugOp** command supports the portation of Microsoft CodeView. For this, command types 0 through 20 correspond approximately to the DOSPTRACE commands that have always been available under OS/2. These functions are equivalent to the ptrace functions that have always been available under Microsoft Xenix.

Below is a list of commands that can be issued using *DebugOp*. Each command type is invoked by changing one or more of the command descriptor fields.

Case	Command Type	Description
0	OpenSession	This command validates *cmd.pid* and *cmd.userNum*. It marks *cmd.userNum* as debuggee and allows further commands on it. This command also controls the level of event notification. The caller can choose to be notified of all events for the debuggee or only those events caused by a DebugOp command. The level of notification is set by *cmd.value*:

Bits	Description
0	Set if notify for all events.
1	Set if notify for only those events explicitly caused by the issuing of a DebugOp command; e.g., SetBreakpoint. (This allows the use of the interactive CTOS debugger on the debuggee as well as an application debugger.)

Case	Command Type	Description	
		The contents of the command buffer are:	
		cmd.userNum	The user number of the debuggee.
		cmd.pid	N/A
		cmd.ty	0
		cmd.value	The notification level, which is described above.
		cmd.ra	N/A
		cmd.sn	N/A
		rq.pReqBuf	N/A
		rq.cbReqBuf	N/A
		rq.pRespBuf	N/A
		rq.cbRespBuf	N/A
1	ReadMem	This command reads *cbRespBuf* bytes from *cmd.sn:cmd.ra* to *RespBuf*.	
		The contents of the command buffer are:	
		cmd.userNum	The user number of the debuggee.
		cmd.pid	The process id of the debuggee.
		cmd.ty	1
		cmd.value	N/A
		cmd.ra	The offset of memory to be read.
		cmd.sn	The selector of memory to be read.
		rq.pReqBuf	N/A
		rq.cbReqBuf	N/A
		rq.pRespBuf	The address of the buffer that memory is written to.
		rq.cbRespBuf	The size of the buffer that memory is written to.
2	ReadMemD	This command is identical to command type 1 (ReadMem).	

Case	Command Type	Description
3	ReadReg	This command reads the registers of *cmd.pid* into *pbRespBuf*. The format of *RespBuf* is a 386 TSS.
		The contents of the command buffer are:

cmd.userNum	The user number of the debuggee.
cmd.pid	The process id of the debuggee.
cmd.ty	3
cmd.value	N/A
cmd.ra	N/A
cmd.sn	N/A
rq.pReqBuf	N/A
rq.cbReqBuf	N/A
rq.pRespBuf	The address of the buffer that the registers are written to.
rq.cbRespBuf	The size of the buffer that the registers are written to.

Case	Command Type	Description
4	WriteMem	This command writes *cbReqBuf* bytes from *pbReqBuf* to the address in *cmd.sn:cmd.ra*.
		The contents of the command buffer are:

cmd.userNum	The user number of the debuggee.
cmd.pid	The process id of the debuggee.
cmd.ty	4
cmd.value	N/A
cmd.ra	The offset of memory to be read.
cmd.sn	The selector of memory to be read.

Case	Command Type	Description	
		rq.pReqBuf	The address of the buffer from which memory is written.
		rq.cbReqBuf	The size of the buffer from which memory is written.
		rq.pRespBuf	N/A
		rq.cbRespBuf	N/A
5	WriteMemD	This command is identical to command type 4 (WriteMem).	
6	WriteReg	This command writes the registers of *cmd.pid* from *pbReqBuf*. The format of *ReqBuf* is a 386 TSS.	
		The contents of the command buffer are:	
		cmd.userNum	The user number of the debuggee.
		cmd.pid	The process id of the debuggee.
		cmd.ty	6
		cmd.value	N/A
		cmd.ra	N/A
		cmd.sn	N/A
		rq.pReqBuf	The address of the buffer the registers are written from.
		rq.cbReqBuf	The size of the buffer the registers are written from.
		rq.pRespBuf	N/A
		rq.cbRespBuf	N/A

Case	Command Type	Description
7	GO/WaitForEvent	This command resumes all nonfrozen pids of *cmd.userNum*. It returns control to the user when the next debugger event for *cmd.userNum* occurs. The contents of the command buffer are:

cmd.userNum	The user number of the debuggee.
cmd.pid	The process id of the debuggee.
cmd.ty	7
cmd.value	N/A
cmd.ra	N/A
cmd.sn	N/A
rq.pReqBuf	N/A
rq.cbReqBuf	N/A
rq.pRespBuf	N/A
rq.cbRespBuf	N/A

Case	Command Type	Description
8	Terminate	This command terminates *cmd.userNum* with the status code specified in *cmd.value*. The contents of the command buffer are:

cmd.userNum	The user number of the debuggee.
cmd.pid	The process id of the debuggee.
cmd.ty	8
cmd.value	The error code.
cmd.ra	N/A
cmd.sn	N/A
rq.pReqBuf	N/A
rq.cbReqBuf	N/A
rq.pRespBuf	N/A
rq.cbRespBuf	N/A

Case	Command Type	Description
9	SingleStep	This command single steps *cmd.pid*.

The contents of the command buffer are:

cmd.userNum	The user number of the debuggee.
cmd.pid	The process id of the debuggee.
cmd.ty	9
cmd.value	N/A
cmd.ra	N/A
cmd.sn	N/A
rq.pReqBuf	N/A
rq.cbReqBuf	N/A
rq.pRespBuf	N/A
rq.cbRespBuf	N/A

Case	Command Type	Description
10	Stop	This command stops all process ids of *cmd.userNum*.

The contents of the command buffer are:

cmd.userNum	The user number of the debuggee.
cmd.pid	N/A
cmd.ty	10
cmd.value	N/A
cmd.ra	N/A
cmd.sn	N/A
rq.pReqBuf	N/A
rq.cbReqBuf	N/A
rq.pRespBuf	N/A
rq.cbRespBuf	N/A

Case	Command Type	Description
11	Freeze	This command suspends *cmd.pid*. If *cmd.pid* is equal to 0, all pids of *cmd.userNum* are suspended (equivalent to Stop).

The contents of the command buffer are:

cmd.userNum	The user number of the debuggee.
cmd.pid	The process id of the debuggee.
cmd.ty	11
cmd.value	N/A
cmd.ra	N/A
cmd.sn	N/A
rq.pReqBuf	N/A
rq.cbReqBuf	N/A
rq.pRespBuf	N/A
rq.cbRespBuf	N/A

Case	Command Type	Description
12	Resume	This command resumes *cmd.pid*. If *cmd.pid* is equal to 0, all pids of *cmd.userNum* are resumed.

The contents of the command buffer are:

cmd.userNum	The user number of the debuggee.
cmd.pid	The process id of the debuggee.
cmd.ty	12
cmd.value	N/A
cmd.ra	N/A
cmd.sn	N/A
rq.pReqBuf	N/A
rq.cbReqBuf	N/A
rq.pRespBuf	N/A
rq.cbRespBuf	N/A

Case	Command Type	Description
13	NumToSel	This command converts link-time segment number, *cmd.value*, to run-time selector (placed in *pbRespBuf*). The contents of the command buffer are: *cmd.userNum* — The user number of the debuggee. *cmd.pid* — The process id of the debuggee. *cmd.ty* — 13 *cmd.value* — The value of link-time selector. *cmd.ra* — N/A *cmd.sn* — N/A *rq.pReqBuf* — N/A *rq.cbReqBuf* — N/A *rq.pRespBuf* — The address of the word that the run-time selector is written to. *rq.cbRespBuf* — The size of the word that the run-time selector is written to (2).
14	GetFpRegs	Not implemented.
15	SetFpRegs	Not implemented.
16	GetLibName	For the DLL handle in cmd.value, this command returns *sbLibSpec* in *pbRespBuf*. The contents of the command buffer are: *cmd.userNum* — The user number of the debuggee. *cmd.pid* — The process id of the debuggee. *cmd.ty* — 16

Case	Command Type	Description	
		cmd.value	The value of Lib Handle.
		cmd.ra	N/A
		cmd.sn	N/A
		rq.pReqBuf	N/A
		rq.cbReqBuf	N/A
		rq.pRespBuf	The address of the buffer that lib name is written to.
		rq.cbRespBuf	The size of buffer that lib name is written to.
17	PidStatus	For *cmd.pid*, this command returns the TStat structure in *pbRespBuf*. If *cmd.pid* is equal to 0, the first pid of *userNum* is returned. The contents of the command buffer are:	
		cmd.userNum	The user number of the debuggee.
		cmd.pid	The process id of the debuggee.
		cmd.ty	17
		cmd.value	N/A
		cmd.ra	N/A
		cmd.sn	N/A
		rq.pReqBuf	N/A
		rq.cbReqBuf	N/A
		rq.pRespBuf	The address of the buffer that pidStatus is written to.
		rq.cbRespBuf	The size of the buffer that pidStatus is written to.
		The TStat format is as follows:	

Offset	Field	Size	Description
0	dbgState	1	Debugger state of the process as follows: Bit 0 Set if debuggee is suspended. Bits 1–7 Reserved.
1	tStat	1	Execution state of the process as follows: 0 Ready to run 1 Suspended 2 Blocked 3 Blocked on critical section semaphore
2	priority	2	Process priority.
4	pidNext	2	Process id of the next process of the same user.
6	pid	2	Process id of this process.

Case	Command Type	Description
18	MapRoAlias	This command modifies the selector in *cmd.value* to alias the selector *cmd.sn.* If *cmd.value* is equal to 0, it creates a new selector. Selector is returned in *pbRespBuf.*

The contents of the command buffer are:

cmd.userNum	The user number of the debuggee.
cmd.pid	The process id of the debuggee.
cmd.ty	18
cmd.value	If zero, a new alias is created and placed here. If nonzero, *cmd.value* is an alias to be remade.
cmd.ra	N/A
cmd.sn	The selector to be aliased.
rq.pReqBuf	N/A
rq.cbReqBuf	N/A
rq.pRespBuf	N/A
rq.cbRespBuf	N/A

Case	Command Type	Description
19	MapRwAlias	This command is identical to command type 18 (MapRoAlias), except the alias is created with writeable access.

The contents of the command buffer are:

cmd.userNum	The user number of the debuggee.
cmd.pid	The process id of the debuggee.
cmd.ty	19
cmd.value	If zero, a new alias is created and placed here. If nonzero, *cmd.value* is an alias to be remade.
cmd.ra	N/A
cmd.sn	The selector to be aliased.
rq.pReqBuf	N/A
rq.cbReqBuf	N/A
rq.pRespBuf	N/A
rq.cbRespBuf	N/A

Case	Command Type	Description
20	UnMapAlias	This command frees the alias in *cmd.value*.

The contents of the command buffer are:

cmd.userNum	The user number of the debuggee.
cmd.pid	The process id of the debuggee.
cmd.ty	20
cmd.value	The alias to be unmapped.
cmd.ra	N/A
cmd.sn	N/A
rq.pReqBuf	N/A
rq.cbReqBuf	N/A
rq.pRespBuf	N/A
rq.cbRespBuf	N/A

Case	Command Type	Description
21	SetBreakpoint	This command sets a breakpoint in *cmd.pid* at the address specified by *cmd.sn:cmd.ra*. *Cmd.value* controls the type of breakpoint and the action taken when the breakpoint occurs. A handle is returned in *pbRespBuf*.

The contents of the command buffer are:

cmd.userNum	The user number of the debuggee.
cmd.pid	The process id of the debuggee.
cmd.ty	21
cmd.value	Flags word.
cmd.ra	The offset of the segmented breakpoint address, or the low word of the linear breakpoint address, depending on the flags word in *cmd.value*.
cmd.sn	The selector of the segmented breakpoint address, or the high word of the linear breakpoint address, depending on flags word in *cmd.value*.
rq.pReqBuf	N/A
rq.cbReqBuf	N/A
rq.pRespBuf	The address of the word where the breakpoint handle is stored.
rq.cbRespBuf	The size of the word where the breakpoint handle is stored (i.e., 2).

Case	Command Type	Description
		The *cmd.value* bits are as follows:

Bits	Description
0–2	The size of the breakpoint. Valid values are 0, 1, 2, 4. 0 denotes code breakpoint; 1, 2, or 4 denotes data breakpoint.
3	Set if write breakpoint; reset if read or write. Applies only to data breakpoints.
4	Set if *cmd (sn:)dra* should be interpreted as a linear address. Reset if logical address.
5	Set if notification of breakpoint is suppressed (this is used with command QueryBreakpoint).
6–15	Reserved.

Case	Command Type	Description
22	ClearBreakpoint	This command removes the breakpoint in *cmd.pid* specified by the handle in *cmd.value*.
		The contents of the command buffer are:

cmd.userNum	The user number of the debuggee.
cmd.pid	The process id of the debuggee.
cmd.ty	22
cmd.value	The breakpoint handle.
cmd.ra	N/A
cmd.sn	N/A
rq.pReqBuf	N/A
rq.cbReqBuf	N/A
rq.pRespBuf	N/A
rq.cbRespBuf	N/A

Case	Command Type	Description
23	QueryBreakpoint	For the breakpoint handle in *cmd.value*, this command returns the breakpoint status in *pbRespBuf*:

Offset	Field	Length
0	*cbStatus*	word
2	*cBreaks*	word
4	*ra*	dword
8	*sn*	word
10	*pid*	word
12	*userNum*	word

The contents of the command buffer are:

cmd.userNum	The user number of the debuggee.
cmd.pid	The process id of the debuggee.
cmd.ty	23
cmd.value	The breakpoint handle.
cmd.ra	N/A
cmd.sn	N/A
rq.pReqBuf	N/A
rq.cbReqBuf	N/A
rq.pRespBuf	The address of the buffer where the breakpoint status is stored.
rq.cbRespBuf	The size of the buffer where the breakpoint status is stored.

Case	Command Type	Description
24	ResetBreakpoint	For the breakpoint handle in *cmd.value*, this command clears the watchpoint count.

The contents of the command buffer are:

cmd.userNum	The user number of the debuggee.
cmd.pid	The process id of the debuggee.
cmd.ty	24

Case	Command Type	Description	
		cmd.value	The breakpoint handle.
		cmd.ra	N/A
		cmd.sn	N/A
		rq.pReqBuf	N/A
		rq.cbReqBuf	N/A
		rq.pRespBuf	N/A
		rq.cbRespBuf	N/A
25	SelToNum	For the run-time selector in *cmd.sn*, this command returns the link-time segment number in *pbRespBuf*. The contents of the command buffer are:	
		cmd.userNum	The user number of the debuggee.
		cmd.pid	The process id of the debuggee.
		cmd.ty	25
		cmd.value	The selector to be mapped.
		cmd.ra	N/A
		cmd.sn	N/A
		rq.pReqBuf	N/A
		rq.cbReqBuf	N/A
		rq.pRespBuf	The address of the buffer where the mapped selector is stored.
		rq.cbRespBuf	The size of the buffer where the mapped selector is stored.

Caution

Breakpoints that are set via DebugOp and are executed with interrupts disabled are proceeded automatically by the Debugger since the Debugger cannot propagate the disabled state back to the caller. If interrupts-disabled breakpointing is required, you must use the Debugger.

Event Notification

When **GO** is pressed, the request is responded to when a debugger event occurs for the user specified in *cmd.userNum*. Upon return from the **GO** command, the contents of *RespBuf* are an Event Descriptor:

Offset	Field	Length
0	*userNum*	word
2	*pid*	word
4	*tyEvent*	word
6	*value*	word
8	*ra*	dword
10	*sn*	word

The event types are as follows:

Event	Type	Description
0	Break	A breakpoint occurred. *Event.value* contains the breakpoint handle. *Event.pid*, *event.userNum*, *event.ra*, *event.sn* are updated. The breakpoint handle is placed in *event.value*. The breakpoint *userNum* is frozen.
1	Death	*Event.userNum* is about to die.
2	Fault	A fault has occurred. *Event.value* contains the type of fault. *Event.userNum*, *event.pid*, *event.ra*, and *event.sn* are updated.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlinfo	1	6
1	rtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	465
12	reserved	6	
18	pCmd	4	
22	cbCmd	2	
24	pReqBuf	4	
28	cbReqBuf	2	
30	pRespBuf	4	
32	cbRespBuf	2	

Error Codes

1000 – 1003	Reserved for future use.
1004	Debugger: Breakpoint already exists. The breakpoint specified in the DebugOp SetBreakpoint command already exists.
1005	Debugger: Internal consistency. An internal error has occurred. To report the error, contact Techical Support.
1006 – 1016	Reserved for future use.
1017	Debugger: Memory not existent. The memory specified by a DebugOp command does not exist.
1018	Debugger: Too many breakpoints. The maximum number of code breakpoints is 16; the maximum number of data breakpoints is 4.
1019 – 1020	Reserved for future use.
1021	Debugger: No such breakpoint. The breakpoint specified in the ClearBreakpoint, QueryBreakpoint, or ResetBreakpoint DebugOp command does not exist.
1022 – 1035	Reserved for future use.
1036	Debugger: Invalid TSS. The debugger encountered a TSS which is invalid. This usually represents an internal error in the Debugger. To report the error, contact Tech Support.

1037	Reserved for future use.
1038	Debugger: Invalid selector. The selector specifed in a DebugOp command is invalid.
1039	Reserved for future use.
1040	Debugger: Invalid process. The process specifed in a DebugOp command is invalid.
1041 – 1046	Reserved for future use.
1047	Debugger: Invalid command type. The command type specifed in a DebugOp command is invalid.
1048	Debugger: Buffer size too small. The buffer size specified in a DebugOp request is too small to hold the requested data.
1049	Debugger: Too many DebugOp requests. There are too many DebugOp requests outstanding.
1050	Debugger: No such session. A DebugOp session has not been opened for the specified client.
1051	Debugger: Session already open. A DebugOp session is already open for the specified client.

Appendix J
Debugging on a Second Monitor

What is Debugging on a Second Monitor?

By adding a particular processor, graphics module, and monitor to your current system, you can use the second monitor as your debugging screen. The application is displayed on the main monitor, while you enter and exit the debugger on the second screen.

To use a second monitor, you must have a processor with extended video support and a graphics controller module.

The supported processors are 80286 and 80386 processors with extended video capability: B28-EV, B28-EXP, B38-EV, B38-EXP, B38-GXP, B38-GXC, B28-CPU, and B38-CPU.

The supported graphics modules are: B25-VG2, B25VG4 (GC-004). The graphics module must be in the configuration as the first module after the CPU. The main monitor attaches to this module.

The supported monitors are: B25-P1 (VM-001) and B25-P2 (VM-002). Either monitor can be attached to the CPU monitor plug. This is the screen that is used for debugging.

To activate second screen debugging, edit the *Config.sys* file to include the :VGACharMapDebugger: option.

For more information about this and other configuration options, see Appendix F, "Configuration Options for the Debugger."

Glossary

A

Address expression.

An address expression is a description of a location in memory. The description consists of one or more symbols, or an indexed or nonindexed parameter.

Application process.

An application process in one that is terminated when the user calls Exit.

B

Breakpoint.

A breakpoint is a user-defined point in the code for a process. Execution stops when a process reaches a breakpoint.

Byte pattern.

A byte pattern is a user-defined group of byte specifiers. The specifiers are separated by commas and enclosed in double quotation marks.

Byte specifier.

A byte specifier is a sequence of two-digit hexadecimal numbers, or a string of characters enclosed in single quotation marks.

C

Clear.

This means to remove a breakpoint from a particular location in memory.

Code listing.

A code listing is an english-language display of code generated by a compiler or translator.

Crash dump.

A crash dump is the output (memory dump) caused by a system failure.

Current process.

This is the process identified in the Debugger by the Debugger internal register (PR). Any registers that are read or written by the Debugger are for the current process.

Current value.

The current value is the value most recently typed by the user, or the value most recently displayed by the Debugger.

E

Echo.

This is the repetition on a parallel printer or a screen of instructions entered by the user and/or material displayed by the Debugger.

Exchange.

This is the path on which a process waits for or receives messages or communications from another process or processes.

I

Indexed address.

An indexed address is the address expression that uses index registers.

Interrupt mode.

This is a debugger operating mode used to debug interrupt handlers or to set breakpoints in the operating system Kernel.

L

Link word.

A link word is a 16-bit address pointing to the next block of data.

Linked-list data structure.

This is a data structure containing elements that are linked by 16-bit addresses (link words) or by 32-bit addresses (link pointers). The **CODE-N** command uses link words.

Linker.

This is a software system that loads and connects together the object programs output separately by a compiler or assembler and, from them, produces a run file.

M

Multiprocess mode.

This is a debugger operating mode used in debugging an application that involves more than one process and that depends on continuous execution of all processes except the ones stopped at breakpoints.

O

Offset.

An offset is the number of bytes by which a memory location is distant from the beginning of a segment.

Output radix.

The output radix is a base of the notation in which Debugger output is expressed (binary, decimal, hexadecimal, or any other base from 2 to 16, inclusive).

P

Parameter.

A parameter is a constant (number, port, or text), a symbol, or one or more unary or binary operators, address expressions, or symbolic instructions.

Physical address.

This is an address that does not specify a segment base, and is relative to memory location 0.

Pointer.

See **Segmented address.**

Port constant.

This is a number followed by an i or an o (indicating an input port or an output port, respectively).

Processor Ordinal Number.

A processor ordinal number is the processor's position (starting with 0) within the shared resource processor. (For an illustration of processor ordinal numbering, see the *CTOS System Administration Guide.*)

Public procedure.

This is a procedure whose address can be referenced by a module other than the module in which the procedure is defined.

Public symbol.

A public symbol is an ASCII character string associated with a public variable, a public value, or a public procedure.

Public value.

This is a value whose address can be referenced by a module other than the module in which the procedure is defined.

Public variable.

This is a variable whose address can be referenced by a module other than the module in which the variable is defined.

R

Register mnemonic.

A register mnemonic is a two-letter symbolic name for a register in the processor (for example, AX, BL, SI).

Release Documentation.

Release Documentation includes Software Release Announcements and the CTOS System Software Installation Planning Guide.

Run file.

This is a file created by the Linker. The run file contains the initial image of code and data for a program.

Run file checksum word.

A run file checksum word is a number produced by the summation of words in a run file. Used to check the validity of the run file.

S

Segment.

A segment is a discrete portion of memory, of a procedure, or of a program.

Segment address.

A segment address is an address of a segment base. For a real mode Intel microprocessor, a segment address refers to a paragraph number (16 bytes). For a protected mode microprocessor, a segment address is an index to a descriptor table containing the segment base address.

Segmented address (pointer).

A segmented address is an address that specifies both a segment base and an offset. Segmented addresses are of the form SA:RA, where SA is the segment address, and RA is the offset.

Segment override.

This is an operating code that causes the microprocessor to use the segment register specified by the prefix when executing an instruction, instead of the segment register that it would normally use.

Set.

A set is a place a breakpoint at a particular location in memory or assign a process identification number to the Debugger internal register (PR).

Simple mode.

This is a debugger operating mode used in debugging a single-process application, for example, a compiled BASIC program.

Stack.

The stack is a region of memory accessible from one end by means of a stack pointer.

Stack frame.

The stack frame is a region of a stack corresponding to the dynamic invocation of a procedure. Consists of procedural parameters, a return address, a saved-frame pointer, and local or temporary variables.

Stack pointer.

The stack pointer is the indicator to the top of a stack. The stack pointer is stored in the registers SS:SP.

Stack trace.

The stack strace is a debugger display of a stack, organized by stack frame.

State variable.

A state variable is a symbolic name of a register that contains data indicating the state of the Debugger (for example, PR, IP, or FL).

Symbol.

This is a sequence of alphanumeric and other characters (underscore, period, dollar sign, pound sign, or exclamation mark).

Symbolic instructions.

Symbolic instructions contain symbols, that is, mnemonic characters corresponding to the assembly language instructions. (These instructions cannot contain user-defined public symbols.)

System process.

This is any process that is not terminated when the user calls Exit.

T

Text constant.

A text constant is a sequence of characters enclosed by quotation marks.

U

User process.

See **Application process.**

Index

UNISYS **Help Us To Help You**

Publication Title

Form Number

Unisys Corporation is interested in your comments and suggestions reguarding this manual. We will use them to improve the quality of your Product Information. Please check type of suggestion:

☐ Addition ☐ Deletion ☐ Revision ☐ Error

Comments:

Name		Telephone number ( )
Title	Company	
Address		
City	State	Zip code

Cut along dotted line

Tape

Please Do Not Staple

Tape

Fold Here

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 817 DETROIT, MI

POSTAGE WILL BE PAID BY ADDRESSEE

UNISYS CORPORATION
PRODUCT INFORMATION
MS 18-007
2700 NORTH FIRST STREET
SAN JOSE, CA 95134-2028

UNISYS

Product Information Announcement

● New Release ○ Revision ○ Update ○ New Mail Code

Title
CTOS® Debugger User's Guide

This Product Information Announcement announces the release and availability of the *CTOS Debugger User's Guide*, dated August 1992. Information in this document is relative to CTOS I 3.4, CTOS II 3.4, CTOS/XE 3.4, and CTOS III 1.0.

This guide describes how to use the CTOS Debugger to debug programs on real and protected mode operating systems. The guide provides hands-on exercises in using the commands as well as debugging tips. The guide is available separately or with binder, slipcase, and training diskette. The diskette is not available separately.

Please address all technical communications relative to this document to
UNISYS, Multimedia Publishing M/S 18–007, 2700 North First Street,
San Jose, CA 95134–2028.

To order additional copies of this document

- United States customers call Unisys Direct at 1–800–448–1424.
- All other customers, contact your Unisys Sales Office.
- Unisys personnel use the Electronic Literature Ordering (ELO) system

Contact your Unisys representative for pricing information.

4359 4977–000 Document only
4357 9523–100 Document with binder and slipcase

Announcement only:
SA, SN, SU, SW, PR5

Announcement and attachments:

System: CTOS
Release
CTOS III 1.0
CTOS II 3.4
CTOS I 3.4
CTOS/XE 3.4
Development Utilities 12.2

Part Number: 4359 4977–000

UNISYS

New Feature Notes

The attached pages amend the following:

CTOS Debugger User's Guide
4359 4977-000

This errata documents new features added to the CTOS Debugger for CTOS III 1.1. You can add this section to the binder you already use for the current version of the *CTOS Debugger User's Guide.*

Distribution lists:
SA, SN, SU, PR5

System: CTOS
Release CTOS III 1.1

Part Number: 4359 4977-001

Section 1
New Debugger Features

This section describes the features available with the newest CTOS Debugger. With these new features you can

- See the contents of data structures displayed in a format you define
  CODE-]
- List currently open files
  CODE-K
- Display exchanges only
  -1 CODE-S
- Display exchanges that contain messages only
  -2 CODE-S
- Translate a linear or logical address to a physical address and see the lock counts and user number associated with it
  CODE-
- Substitute the !, @, #, $, or % characters for the the wildcard characters * and ?
  CODE-F

The Debugger also disassembles all 386 instructions including 32-bit instructions and provides support for case-sensitive symbol files.

On CTOS III 1.1 you also can

- Debug on multiprocessor EISA systems
  CODE-M
- Query the state of the floating point unit
  CODE-^
- Set breakpoints in 32-bit code segments

Displaying the Contents of Data Structures

You can use the **CODE-]** command to display the contents of a data structure. You provide a definition of the format for the structure in the file *[Sys]<Sys>DebuggerStruct.txt*; the Debugger uses this specification to find the information you need and display it for you in the format you defined. You can define formats for single structures and linked structures. Nested structures are not supported.

The default copy of *[Sys]<Sys>DebuggerStruct.txt* distributed with the operating system defines formats for some operating system data structures. If you want to add formats to this file you can use the Editor at any time to make changes to the file. As soon as you close the file again and save you can use the new formats.

Defining a Format

Figure 1-1 shows a sample format descriptor for the ExParDesc data structure as it appears in the file *[Sys]<Sys>DebuggerStruct.txt.* (ExParDesc is an operating system data structure that is defined in the *CTOS Procedural Interface Reference Manual.*)

Figure 1-1. Sample Format Description from *DebuggerStruct.txt*

```
;
; ExParDesc
;
epd ('    currRunFile    ' 0/s '
    exitRunFile    ' 4f/s '
    pCharMap       ' 0DD/p '
    sCharMap       ' 0E1/w '
    pVcb           ' 0E3/p '
    prgpVidLine    ' 0EB/p '
    pKbdBuf        ' 0AC/p '
    fReadOrPeek    ' 185./b '
    pPntDevData    ' 0EF/p '
    fActionFinishDisabled ' 0BA/b
)
```

The exact format for a format description is defined in Table 1-1, later in this subsection.

The general format for the format description of a single data structure is

```
FormatName (String or FieldDescriptor, NextString or
        FieldDescriptor)
```

You can include format descriptions for as many data structures as you want in *DebuggerStruct.txt*. There is no additional delimiter required between format descriptions.

All text on each line that begins with a semi-colon (;) is treated as comment text. Notice that in Figure 1-1, the the full name of the structure is included as a comment before the format description, and that the actual description abbreviates the structure name.

Strings are enclosed in single quotes. Note that in the example shown in Figure 1-1, the input has been set up so that the quotes that surround the string that describes the field name include a line-feed character (**SHIFT-RETURN**). This causes each new field and the associated data to be displayed on a new line.

Field descriptors include an offset and one of several field type descriptors. The field types are: byte, word, double word, sbString, zbString, integer, pointer, selector, and userNum. Fields can also be marked as links, so that a linked list of structures can be displayed.

Table 1-1. Format Description Fields for *DebuggerStruct.txt*

Field	Description		
FormatName	ASCII text (usually an abbreviation for the structure name)		
String	ASCII text within single quotes (') used for the name of each field in the structure. For example, 'ExitRunFile'		
FieldDescriptor	Offset/FieldType For example, 0DD/p		
	Offset	Hexadecimal or decimal 16-bit number or blank. If blank, the offset is implied by the offset + length of the prior field. The number is hexadecimal unless suffixed by a period (.) which, as with all debugger input denotes a decimal number.	
	FieldType	Integer [*FieldTypeDefinition*]	
		FieldType definitions:	
		s	sbString (first byte is length of string)
		z	zbstring (zero-terminated string)
		a	ASCII byte
		b	binary byte (unsigned 8 bit)
		w	word (unsigned 16 bit)
		i	integer (signed 16-bit)
		q	quad (unsigned 32 bit)
		e	selector
		p	pointer (sn:ra, ra is 16 bits)
		u	usernum (displays partition and run file name)
		x	pointer (displayed symbolically)
		l	length of structure (used to compute address next structure for packed arrays)
		wl	near (16-bit ra) link to next structure
		ql	near long (32-bit ra) link to next structure
		el	selector link to next structure
		pl	pointer (sn:ra, 16-bit ra) link to next structure

Using the CODE-] command

You can use the **CODE-]** command to

- Display all structure format names that match a wildcard string
- Display the memory at a specified logical or linear address in the specified format
- Display the contents of memory in the specified format at the 'next' address (linked data structure or previous structure)
- Display the contents of memory at the specified address in the specified format for a specific number of times

Figure 1-2 shows sample output from **CODE-]** for the format specified as **epd** in the example Figure 1-1 (described earlier).

Figure 1-2. Sample Output from CODE-]

```
 *2558:4a,'epd' ^]
currRunFile  [sys]<sys>exec.run
exitRunfile  [sys]<sys>exec.run
pCharMap     0070:0000
sCharMap     1220
pVcb         2558:05E2
prgpVidLine  2558:02B2
pKbdBuf      2558:06A2
fReadOrPeek  FF
pPntDevData  2558:07F2
fActionFinishDisabled 00
```

To display all format names which match the wild card string, type

'wildcardString' **CODE-]**

Example:

```
 *'*'^]
Ascb asib10 asib11 cpd ced ccb cdt dcb Dct doi epd exUcb fab
fcb fcb2 fib frib icb iccSeg iccBuffer iccRcb iccStack
iccRcbSeg iob Lcb lucb NetServerData nmb parCnfg parDesc pdh
pub puba QuietRq runHdrV6 runHdrV8 scsiIob scat swa scb tcb
trb tss vhb ucb upb Vf vStats XBlock XeIccSeg XeRcb XeRcbSeg
```

To display the memory at the specified address with the format specified, type

addr, formatName **CODE-]**

addr can be logical or linear.

Example:

```
 *0df0:0,'tss'^]
rIOMap=0134 rRbg87=00D6 tyDev=0000 iIntLevel=0000
wDSInt=0000 wFlInt=0000 wIpInt=0000 wCsInt=0000
oPcb=4946 wErrorCode= 0000 laFault=FEF45011 pSem=0000:0000
wOs2Tid=0000 glaTss=00797500 pHeap=0000:0000
wDsIntFltr=0000 wFlIntFltr=0000 wIpIntFltr=0000
wCsIntFltr=0000
cCritSect=0000 cPendingSusp=0000 pKernelStack=22E8:00C0
pUserStack= 0084:FE82
```

To display the contents of a specified data structure starting at a specified address and showing the contents of the structure at a specified number of subsequent occurences (cFormats), type

cFormats, address, formatName **CODE-]**

Note that this is sensible only when links or structure length sizes are used in the format definition.

To simply display the memory at the next address with whatever format was specified previously, just type

CODE-]

The next address is computed from either the *link* field or the previous structure or, in the case of packed arrays, the size and address of the previous structure. See Table 1-1 for the description of the *link* field (field type 'l').

Debugging on a Multiprocessor System

CTOS III 1.1 runs on multiprocessor EISA-bus workstations. With the new Debugger, you can switch back and forth between the master processor (the host) and any of the I/O processors (IOPs). The debugger shows you which processor you are running on. You use all available debugger commands on the IOPs in the same way as on the main processor.

There is actually a separate copy of the Debugger running on each processor. The Debugger on the master processor communicates with the Debuggers on the IOPs over a separate EISA-bus channel, sending keystrokes (commands) to them, capturing their output, and then displaying it on the video display. When you want to examine memory on another board for example, the Debugger on the master processor sends the keystrokes you enter (for example, *10, 84:0* **CODE-D**) to the Debugger executing on the IOP. That Debugger treats this data as command input and executes its memory fetching and displaying operations. The output of the command is directed to the EISA bus, where it is received by the host Debugger and displayed on the video display. Note that this architecture is different than that used on CTOS/XE and provides a more complete Debugger function. Breakpoints and single-stepping, as well as interrupt-disabled debugging, are supported on the IOP.

To debug on a multiprocessor system you need to understand interprocess communications and processor board naming. See the *CTOS Operating System Concepts Manual* and the *CTOS System Administration Guide* for background information.

Using the CODE-M Command

You use the **CODE-M** command to switch back and forth between the Debugger on the main processor and the Debugger on a specified IOP card. Once you make the switch, other Debugger commands operate just as you expect. You can always tell if the Debugger you are using is executing on an IOP because the Processor or IOP name is displayed next to the Debugger prompt.

To display a list of available IOP Cards when you are using the debugger on the main processor, type

CODE-M

The Debugger displays the processor name in front of the Debugger prompt on the screen as shown below:

```
 #^m
name  id
EP01  04
EP02  05
```

To start a debugger on an IOP or to switch to a debugger that is already running on an IOP type

'processorID' or *'processorName'* **CODE-M**

To switch back to the main debugger, type

CODE-M

Note that this does not stop the Debugger on the IOP. You can use **GO** or **CODE-P** on the IOP just as you would on the main processor to resume processing.

Note that you cannot switch from one IOP directly to another. You must return to the Debugger on the main processor first.

The example below shows switching to the debugger on EP01, then back to the main processor:

```
 #4^m
EP01\Debugger x3.5.38_6/5-16:43  (Simple Mode)
EP01\Stopped at 8C:2D in process 13
EP01\ *^m
 #'ep01'^m
EP01\ *
EP01\ #^m
 #
```

Note: *If you use the **Cluster View** utility to access an IOP on the server it is a good idea to access the master processor on the server first, then switch to the IOP. If you access the IOP on the server directly from the client workstation you will not be able to switch to the master processor without stopping your debugging session on the IOP.*

Extensions to CODE-S

The **CODE-S** command is described in detail in Section 7, "Display Commands," in the *CTOS Debugger User's Guide*.

There are two new extensions to the **CODE-S** command. They allow you to display exchange information only and to display only exchanges which contain messages.

'-1, CODE-S ' displays exchange information only, without process information.

Example:

```
 *-1^s
003 SysIn                       Processes   |03|
006 Sched                       Processes   |07|
007 exchQuiet                   Processes   |06|
008 MassIo                      Processes   |09|
00A LclFileSys                  Processes   |08|
00B FileSys                     Processes   |02|
00C Agent                       Processes   |0A|
010exchRqTracker Messages
|2558:912(0,8A00)|3280:FE9A(5,8800)|
011 MstrAgentRcv                Processes   |0B|
014 exchPit                     Processes   |01|
01B HwId                        Processes   |05|
028 OS                          Processes   |0C|
02B OS                       Messages    |01:F3|
02D PS_V                        Processes   |5E|
035 Vdm_Vga   4.2               Processes   |0F|
037 Vdm_Vga   4.2               Processes   |11|
039 IkServer                    Processes   |12|
03A Primary                    Processes   |13*|
```

'-2, CODE-S ' displays only exchanges which contain messages.

Example:

```
 *-2^s
010 exchRqTracker    Messages     |2558:912(0,8A00)|3280:FE9A(5,880
02B OS                Messages     |01:F3|
```

These new features are available on any version of CTOS.

For CTOS III 1.1 only, the Debugger also now shows messages which are termination requests with an asterisk ('*') on the **CODE-S** display. This is helpful for debugging hangs resulting from improper handling of termination requests.

Example:

```
010 exchRqTracker       Messages    |*2588:20(0,8A00)|
```

This feature is available only when running on CTOS III 1.1.

Querying the Floating Point Unit State

With the CTOS III Debugger you can now query the state of the Floating Point Unit (FPU). To do so you use the command **CODE-^**.

The command **CODE-^** displays the FPU state of the current process register (PR). The debugger displays both raw and formatted FPU control, status, and tag words, as well as the used portion of the FPU stack.

Example:

```
 *^^
control=1370 (UM PM 64bit RoundNear)
status=0800 (TOP=01)
tag=0003 cs:ip=009C:0024 pOp=0284:0060 stack:
0:+ 2E0008 * ACD6C90000000000 (0000000000C9D6AC0740)
1:+ 2E0000 * 9000000000000000 (000000000000090FF3F)
2:+ 2E0003 * C000000000000000 (00000000000000C00240)
3:- 2E0004 * 8000000000000000 (000000000000008003C0)
4:+ 2E0004 * 8000000000000000 (00000000000000800340)
5:+ 2E0001 * 8000000000000000 (00000000000000800040)
6:+ 2E0000 * 8000000000000000 (0000000000000080FF3F)
```

Note that the DebugOp API now supports case 14 (GetFpRegs) and 15 (SetFpRegs). In both cases, the 32-bit format of the FPU is used (see the *i486 MicroProcessor Reference Manual*).

Listing Open Files

You can use the new command **CODE-K** to display a list of files which are currently open. It displays the status, handle, user number, and name of each open file.

You can use **CODE-K** with no parameters to display the open files on the *[Sys]* volume or you can type a device specification or volume name followed by **CODE-K**.

Querying Page Frame Statistics

A new option,the **CODE-\** command allows you to query page frame statistics. A linear or logical address may be argued to **CODE-\** to display the physical address, lock counts, and user number of the memory. The command is useful for troubleshooting DMA programming.

Example:

```
 *8c:8^\
pa = 00102000

cRqLocks=0003 cWaitFaultLocks=0000 cPageLocks=0000
cDmaLocks=0000
userNum=0005
```

Changing Symbol WildCard Characters

If you need to, you can change the characters used as wildcard characters used in symbol name matching from the question mark (?) and the asterisk (*) to alternate characters. This supports compilers such as Microsoft C 7.0 which produce symbolic addresses with embedded question mark and/or askerisk characters.

The alternate characters used must be one of the following set:

? * ! @ # $ %

-1,'b' <CODE>-F
This sets the question mark wildcard character to 'b'.

-2,'b' <CODE>-F
This sets the asterisk wildcard character to 'b'.

43594977-001