Burroughs

XE 500 BTOS Debugger

Operations Guide

Relative To Release Level 6.0
Priced Item
November 1986

Distribution Code SA
Printed in U S America
1207750

Burroughs

XE 500 BTOS Debugger

Operations Guide

Relative To Release Level 6.0
Priced Item
November 1986

Distribution Code SA
Printed in U S America
1207750

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

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

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

Correspondence regarding this publication should be forwarded, using the Product Improvement Card at the back of this manual, or remarks may be addressed directly to Burroughs Corporation, Corporate Product Information East, 209 W. Lancaster Ave., Paoli, PA 19301, U.S.A.

About This Guide

Purpose

This manual describes the XE 500 BTOS Debugger and provides the programmer with a listing and explanation of the commands used in program debugging.

Scope

This guide describes the specific features of the XE 500 BTOS Debugger and explains what commands control the Debugger and how to use them. It does not explain how to interpret Debugger responses or offer guidance on how to best use the Debugger.

Audience

This manual is written for experienced programmers with an extensive knowledge of system structures, assembly language, the BTOS assembler, and linker/librarian functions.

Prerequisites

The programmer using this guide should be familiar with system structures and have experience with reading, writing, and interpreting assembly language. Some familiarity with the BTOS Debugger is helpful.

How to Use This Manual

Users who are not familiar with the BTOS Debugger would find it helpful to read this entire manual before trying to use the Debugger. Users who are familiar with the BTOS Debugger should use this manual as a reference guide.

Organization

Section 1: **Overview of the XE 500 BTOS Debugger** introduces the Debugger and tells where to find information concerning operating system generation for the Debugger.

Section 2: **Debugger Concepts** provides an explanation of the command parameters used by the Debugger and how to display the user screen.

Section 3: **Debugger Connection and Activation** explains how to connect a terminal or workstation to the Debugger and how to activate the Debugger. It also explains how to interpret the Debugger prompts and what the modes are for using the Debugger.

Section 4: **Symbolic Values** describes symbols files and how to open them for the Debugger.

Section 5: **Memory and Registers** describes how to examine and modify memory, how to change the open location and change the contents of a memory location, defines the process register, how to examine and modify registers, how to use register mnemonics, and how to display a trace of a stack and the status of processes and exchanges.

Section 6: **Forcing a Crashdumpfile** explains how to force a crashdump on another processor board from the one on which you are currently debugging.

Section 7: **Breakpoints** describes the addresses (breakpoints) that are set for suspending processes by the Debugger and the commands used to control them.

Section 8: **Command Summary** is a listing of all the commands, their parameters, and meanings for both Burroughs workstation and RS232-C terminal users.

Appendix A: **Status Codes** provides a listing of the XE 500 BTOS Debugger staus messages.

Appendix B: **Quick Reference to Debugger Commands** is a single page listing of Debugger commands that can be used as a handy reference when actually using the commands.

A glossary and index follow Appendix B.

Conventions Used in This Manual

The examples containing commands and references to commands in the text of this manual are those that would be used on a Burroughs workstation (for example, CODE-X executes and opens the next instruction from the workstation and CONTROL-X executes the same command from an RS232-C serial terminal). A command summary appears in Section 8 which shows the corresponding commands for RS232-C serial terminal users and a quick reference guide appears in Appendix B.

Text in italics represents variables.

All numbers are decimal unless otherwise specified.

Related Product Information

BTOS Assembler Programming Reference Manual

This manual provides descriptive and operational information for the Burroughs assembler and assembly language used in Burroughs BTOS systems applications.

BTOS Linker/Librarian Programming Reference Manual

This manual contains introductory, procedural, and reference information on the Linker and Librarian (Language Development Software).

BTOS Reference Manual, Volumes 1 and 2

This manual is a two volume guide that details the Burroughs operating system (BTOS). It explains the services for managing system processes, messages, memory, exchanges, tasks, video, disk, keyboard, printer, timer, communications, and files.

XE 500 BTOS Programming Reference Manual

This manual describes XE 500 BTOS Programming. It provides information supplemental to the *BTOS Reference Manual, Volumes 1 and 2* concerning Burroughs XE 500 operating system software for systems and applications programming.

XE 500 BTOS Customizer Operations Guide

This manual outlines the procedures necessary to customize an XE 500 BTOS operating system and to generate a Debugger operating system.

XE 500 BTOS Operations Guide

This manual provides basic information required to perform routine operations in the XE 500 BTOS environment. It should be used as a supplement to the *BTOS Standard Software Operations Guide.*

XE 500 BTOS Administration Guide

This manual describes the administrative tasks for the XE 500 BTOS system, such as managing files, system security, archiving, and so on.

XE 500 BTOS Operations Reference Manual

This manual provides a listing and description of all master commands that can be run in XE 500 BTOS.

XE 500 BTOS Installation and Implementation Guide

This manual presents the procedures used to install, configure, and implement the BTOS software on the XE 500.

BTOS Standard Software Operations Guide

The manual presents introductory, procedural, and reference information for using the standard features of BTOS.

Contents

Illustrations

Overview of the XE 500 BTOS Debugger

The XE 500 BTOS Debugger is a programming tool for helping programmers find problems in user or system software. The Debugger is an operating system process with the capability of stopping any other concurrently executing user or system process for the purpose of examining the progress of its execution. The user may subsequently find errors in the code or execution of the code which may cause a process to fail. The Debugger can assemble symbolic user input into binary machine instructions and it can disassemble the contents of memory into assembly language source instructions. It responds to commands as they are entered and stops at specified addresses, called breakpoints, until it receives further commands. The Debugger allows you to examine and modify memory, to set and clear breakpoints, and to produce formatted displays of memory.

Generating a Debugger Operating System

The Debugger is part of the standard software release package; however, to use the Debugger, a separate operating system must be generated containing the Debugger process. Generating a Debugger operating system follows procedures similar to those used for customizing an operating system; see the *XE 500 BTOS Customizer Operations Guide* for specific Debugger operating system generation procedures.

Debugger Concepts

Command Parameters

All Debugger commands have the same format: from zero to two parameters followed by a single command character. When a command has two parameters, they must be separated by a comma. The command characters are:

- Up arrow
- Down arrow
- Left arrow
- Right arrow
- MARK
- RETURN
- =
- The CODE and the alphabetic or symbol key pressed simultaneously.

Note: *The references to commands and examples containing commands in the text of this manual are those that would be used on a Burroughs workstation (for example CODE-X on a workstation is CONTROL-X on an RS232-C serial terminal). A command summary appears in Section 8 showing the corresponding commands, parameters, and meanings for RS232-C serial terminal users.*

The Debugger accepts parameters similar to the parameters permitted in assembly language. These parameters are:

- Constants (numbers, ports, and text).
- Symbols.
- Composite parameters formed with parentheses, the unary operators - and PTR, and the binary operators :, +, -, *, and /.
- Address expressions.
- Symbolic expressions.

Constants: Numbers, Ports, and Text

Numbers

A numeric constant is a sequence of hexadecimal digits (0 through 9 and A through F), ending with an optional period, "." or h. If a numeric constant ends in h or omits the ending, it is a hexadecimal number. A numeric constant can begin with the characters A through F only if it is prefixed by zero. Examples of numeric constants are:

123h A hexadecimal parameter.

123 The same parameter.

123 The decimal number 123.

0AF A hexadecimal parameter with the required prefix of 0.

Ports

A port constant is a number followed by the character "i" or "o." An "i" indicates that the port is a microprocessor hardware input port. An "o" indicates that the port is an output port. Examples of port constants are :

12i is the input port that has port address 12.

0A2o is the ouput port that has the address 0A2.

The Debugger can read and write an input port constant. However, the Debugger can only write an output port constant. The ← and → commands open output ports for modification without reading them.

Text

A text constant is a sequence of characters enclosed in single quotation marks. To include a single quotation mark within a text constant, precede it with another single quotation mark. Text constants of one or two characters can be used anywhere a number is allowed. Text constants of more than two characters can only be used with CODE-F. Examples of text constants are:

'abcd'	A four-character constant.
''a'	A two-character constant (a single quotation mark and a).

Symbols

A symbol is a sequence of alphanumeric characters that cannot begin with digits 0 through 9. There are four kinds of symbols:

- User-defined public symbols in a symbol file produced by the Linker from a source program.
- Standard processor register mnemonics.
- Names of internal state variables of the Debugger.
- The period (.) indicates the currently opened location.

Address Expressions

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

SYM	The simplest address expression is a symbol.
RqInterface = 22C3:0 RqInterface + (100/2) RqInterface = 22C3:80	A more complex expression involving a symbol.
bx B46B [BX + 5]	An indexed parameter.
ES:[BX+5][SI]	A doubly indexed parameter with a segment override prefix.

Symbolic Instructions

Symbolic instructions have the same structure and semantics as in assembly language. Examples of symbolic instructions are:

MOV AX, WORD PTR [BX+5]

LOCK INC [BX]

Debugger Connection and Activation

Connecting a Terminal to the Debugger

There are a number of ways to activate the Debugger, as well as a number of ways to make a connection to the Debugger. Before you can activate the Debugger to debug your programs or operating system, you must connect your terminal or workstation to it. The following is a list of key terms used when connecting your terminal or workstation to the Debugger:

- **Destination Processor:** the processor in which the Debugger will execute. This processor must be executing an operating system that contains the Debugger. (For more information on generating a Debugger operating system, see the *XE 500 BTOS Customizer Operations Guide*.)
- **Debug Console:** a device which allows the operator to provide keyboard input to and view output data from the BTOS Debugger. This can be a BTOS workstation or an RS232-C serial terminal.
- **Associate Processor:** the processor which will provide a logical connection between the Debug Console and the Destination Processor. It cannot be the same processor as the Destination Processor.
- **Direct:** a physical connection from the Debug Console to the Destination Processor.
- **Indirect:** a non-physical connection from the Debug Console to the Destination Processor.

The three methods by which a Debug Console can be connected to a BTOS Debugger are as follows:

1 Direct connection through an RS232-C serial terminal.

2 Indirect connection through an RS232-C serial terminal.

3 Indirect connection through a BTOS workstation.

The procedures for using each of these three methods are outlined in the following subsections.

RS232-C Serial Terminal Direct Connection (Method 1)

This method is only applicable for debugging Cluster Processors (CPs) and Terminal Processors (TPs). Figure 3-1 illustrates this connection.

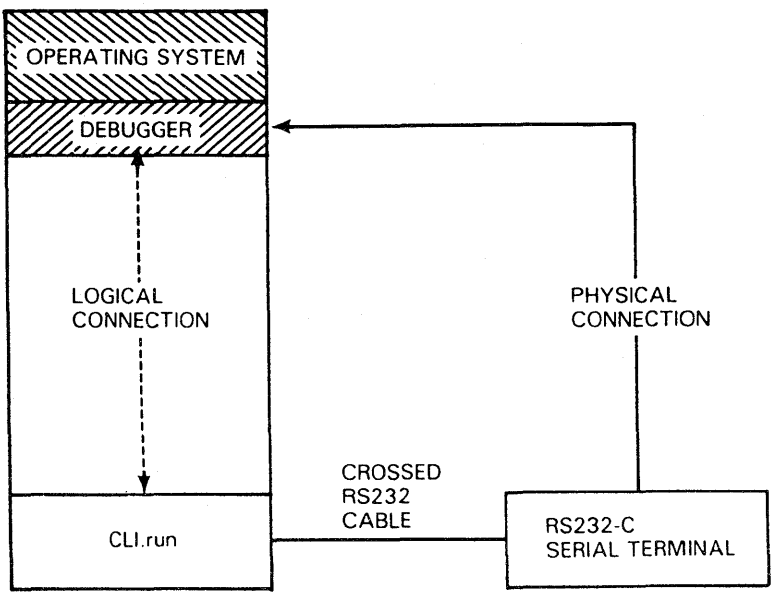
1 Ensure that **CLI.run** is running in the primary partition on the Destination Processor.

2 The Destination Processor's **CP***nn*.cnf or **TP***nn*.cnf configuration file must contain a "Connect=CTOS" parameter for the CLI port. Add this parameter at the *end* of the async*n* line of the configuration file by typing:
,connect=ctos

where async*n* is async3 on a CP and async10 on a TP. If the particular async*n* line is missing, you will have to add it to the paramter list.

3 The RS232-C serial terminal must be connected to the Destination Processor's CLI port through a crossed RS232 cable.

Figure 3-1 **RS232-C Serial Terminal Direct Connection**



E7643

RS232-C Serial Terminal Indirect Connection (Method 2)

This method is applicable to debugging **all** BTOS Processors. The Associate Processor must be a CP or a TP. Figure 3-2 illustrates this connection.

1 Ensure that **CLI.run** is running in the primary partition on the Destination Processor.

2 The Associate Processor's **CP***nn*.cnf or **TP***nn*.cnf configuration file must contain a "Connect=CTOS" parameter for the CLI port. Add this parameter at the *end* of the async*n* line of the configuration file by typing:
,connect=ctos

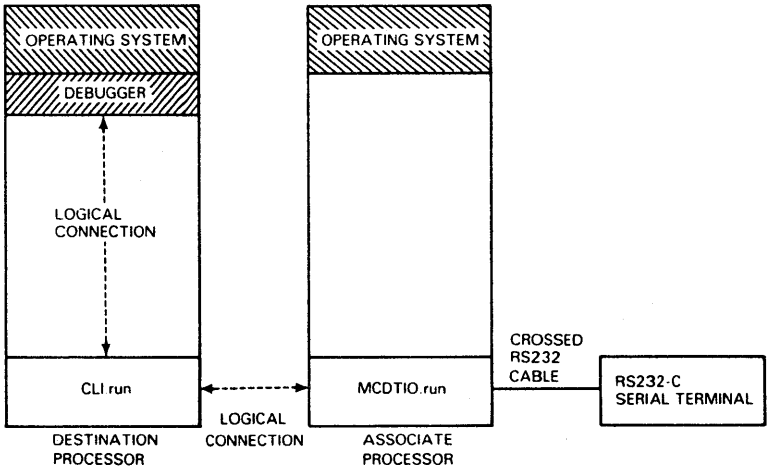
where async*n* is async3 on a CP and async10 on a TP. If the particular async*n* file is missing, you will have to add it to the parameter list.

3 The RS232-C serial terminal must be connected to the Associate Processor's CLI port through a crossed RS-232 cable.

4 Execute the **MCDTIO** command at your terminal. This replaces the **CLI.run** file with the **MCDTIO.run** file in the Associate Processor's primary partition. To execute MCDTIO, type:
run <admin>mcdtio.run, *x*P*nn*

where *x*P*nn* is the Destination Processor. This will return a message that MCDTIO is connected or "talking" to the Destination Processor. Respond with a carriage return.

Figure 3-2 **RS232-C Serial Terminal Indirect Connection**



BTOS Workstation Indirect Connection (Method 3)

This method is applicable to debugging all BTOS processors; it is shown in Figure 3-3. Note that all BTOS workstation connections are indirect and that the Debug Console can be connected to any CP. For this connection, the Associate Processor can be any BTOS Processor.

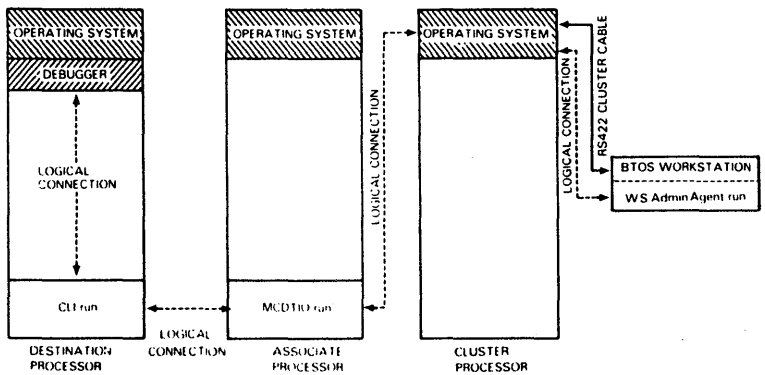
1 Ensure that **CLI.run** is running in the primary partition on the Destination Processor.

2 Ensure that Associate Processor has **MfAdminAgent.run** or **SlvAdminAgent.run** running in the primary partition.

3 Execute the **MCDTIO** command at your workstation. This replaces **MfAdminAgent.run** or **SlvAdminAgent.run** with the **MCDTIO.run** file in the primary partition of the Associate Processor. To run MCDTIO, type:
mcdtio

On the CPU line of the mcdtio menu, enter *x*P*nn*, which is the Destination Processor to which you wish to connect. This will return a message that MCDTIO is connected or "talking" to the Destination Processor. Respond with a carriage return.

Figure 3-3 **BTOS Workstation Indirect Connection**



*NOTE THE CLUSTER PROCESSOR CAN ALSO BE THE ASSOCIATE PROCESSOR

7645

Activating the Debugger

Once your terminal or workstation is connected to the Debugger, you can activate the Debugger in four ways:

1 When an executing process encounters a previously placed breakpoint.

2 When DEB (for debug) is substituted for RUN in the Command Line Interpreter (CLI).

3 When you press the CODE, SHIFT key, and A keys simultaneously (referred to in this manual as CODE–SHIFT-A) on a workstation or CONTROL-A on an RS232-C terminal.

4 When you press the CODE, SHIFT key, and B keys simultaneously (referred to in this manual as CODE–SHIFT-B) on a workstation or CONTROL-B on an RS232-C terminal.

When the Debugger is active, it displays its most recent dialog with you, followed by the Debugger prompt.

When the Debugger is activated because a breakpoint is encountered, it displays a description of the break (the breakpoint address and a corresponding process number), issues the Debugger prompt, and waits for your commands. While the Debugger is waiting, all keyboard input is treated as part of its command input.

Debugger Prompts

When you are using the Debugger, the display screen shows your most recent dialog with the Debugger and it also shows a Debugger prompt. The Debugger prompt is an asterisk (*), a pound sign (#), a space ( ), an exclamation point (!), or a greater than sign (>), depending on the type of debugging you are doing. Their meanings are as follows:

*	The Debugger has suspended the current process.
#	The Debugger has not suspended the current process.
!	The Debugger is at an interrupt level.
>	The system has abnormally terminated.

Exiting the Debugger

To dismiss the Debugger, press the GO or ESCAPE key. The Debugger responds by displaying the message:

Exiting Debugger

All input is now directed to the executing process.

Deactivating the Debugger

CODE-K exits and deactivates the Debugger. After pressing CODE–K twice, you can only reenter the Debugger by rebooting the XE 500.

Debugger Modes

There are three basic modes for using the Debugger: simple, multiple-process, and interrupt.

- *Simple mode* is for debugging a single-process user task and is the most common mode to use.
- *Multiple process mode* is for debugging multiple process user tasks whose functional behavior depends on the continued execution of all processes except those explicitly suspended at breakpoints.
- *Interrupt mode* is for debugging interrupt handlers or debugging that requires the setting of breakpoints in the operating system kernel.

Simple Mode

This mode results from CODE-SHIFT-A or the execution of a CODE-B breakpoint encountered after a CODE-SHIFT-A invocation of the Debugger.

In this mode, all user processes are suspended when you or the program enters the Debugger. The operating system services and interrupt handlers are not affected.

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

Multiple Process Mode

This mode results from CODE-SHIFT-B or the execution of a CODE-B breakpoint encountered after an CODE-SHIFT-B invocation of the Debugger.

User processes that have not executed breakpoints continue concurrent execution; user processes executing CODE-B breakpoints are suspended. The operating system services and interrupt handlers are not affected.

If the current process in multiple-process mode is suspended, the CODE-P command causes that process to resume. However, in multiple-process mode, the CODE-P command does not exit from the Debugger; you must press GO to exit.

The Debugger prompt is the pound sign (#) if the current process, as defined by PR, is executing and the asterisk if it is not executing.

The Debugger goes into interrupt mode whenever it encounters a CODE-I breakpoint.

Interrupt Mode

This mode results from the execution of a CODE-I breakpoint. When a breakpoint is encountered, the Debugger takes control of the interrupt system, freezing the entire state of the processor.

The Debugger prompt is the exclamation point (!).

Symbolic Values

The Linker produces symbol table files that contain the locations of public symbols and line numbers in a program. (See the *BTOS Linker/Librarian Programming Reference Manual* for more information about the Linker.) There is one such symbol file for each run file. You must "open" a symbol file before the Debugger can refer to it. To open a symbol file, type CODE-F with the file name of the symbol file as the parameter. This opens the symbol file for your program and allows you to refer to its public symbols. The Debugger can refer to only one symbol file at a time; when a CODE-F is issued, the Debugger discards any previously opened symbol file.

The Debugger attempts to decode absolute addresses into symbolic names. For example, when an appropriate symbol table is in use, the instruction:

CALL 0FFEF:336

might appear as:

CALL ErrorExit

To suppress symbolic output, type:

CODE-F

and the message:

Symbols off

appears.

To re-enable symbolic output, type:

CODE-F

again and the message:

Symbols on

appears.

CODE-F suppresses only symbolic output. Symbolic names can be used as input any time a symbol file is in use.

There are three ways to specify the offset when you are loading symbol files:

- If you are debugging an operating system process, the offset is 0.
- If you are debugging in the primary partition, no offset is specified.
- If you are debugging in a background partition:

1 Run MPartition Status with verbose to get SaLowBound for the Partition.

2 SaLowBound:0 is the pointer to the Partition Configuration Block (the third two byte entry is the offset of the ASCB).

3 SaLowBound:oASCB = 18h is oLastTask (the number which goes into the symbol file).

4 oLastTask, 'Symbol File' CODE-F.

For example, type:

1234, 'Fred.sym'

to debug a program whose base address is hexadecimal 1234 (paragraph hex 1234). You do not need to specify the program base address when debugging a primary task, since the Debugger can determine it.

Memory and Registers

Examining and Modifying Memory

Examining Memory

When the Debugger is active, you can examine memory by typing a parameter that designates a machine address, followed by one of the commands, left arrow, right arrow, or MARK. The Debugger displays the contents of the designated address and opens it for modification; this is known as the *open location*.

To display a single byte, type:

*addr*left arrow

To display a single word, type:

*right*arrow

To display a symbolic instruction, type:

*addr*MARK

Some examples are:

AL ← 1F one byte

DS:100 → 1F20 one word

CreateISAM +10 ▲ MOV AX, WORD PTR [BX]
one instruction

The MARK key echoes as ▲ . You can also use the memory examination commands to display several locations by preceding the parameter with a repeat number. The specified number of parameters are displayed and the last of them remains open. For example, to see the next three words, type:

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

The Debugger always prompts when it is ready for input. How it prompts depends on the location:

- If there is no open location (which happens when the Debugger was just activated or if the previous command closed all open locations), then a Debugger prompt is given.
- If there is an open location, then the prompt is a space.

Changing the Contents of a Memory Location

When the Debugger prompts with a space, you can type in a new value to change the contents of the open location to any value. For example, in response to the DS:101 prompt to change the contents DS:101 to 2F37, type:

```
DS:101
DS:101 ——→ 2F30
```

Remember that instructions in the processor vary in length. The entry of an instruction can modify bytes of memory beyond those of the instruction previously displayed or can leave some final bytes of the original instruction dangling after the end of the new instruction.

Changing the Open Location

The open location is changed in each memory modification command, according to the command. There are three commands:

- RETURN: the formerly open location is closed and no new location is opened.
- Up arrow: the previous location is opened.
- Down arrow: the next location is opened.

Process Register

The Debugger internal register, PR, always identifies the current process. PR is automatically set to the process number of the process that most recently encountered a breakpoint. When you invoke the Debugger just before the execution of an application system begins, PR is set to the number of the initial user process. It should be noted that PR is not an actual register, but is a pointer to an address for the Debugger.

All commands concerned with processes refer implicitly to the current process. To debug a single process program, you do not need to refer to PR. To debug a multiple-process program, you must be aware of which process is current.

Current Value

The Debugger retains a special value known as the current value. The current value is either the value most recently displayed by the Debugger or the most recent value typed by you, whichever occurs last. You can type = to redisplay the current value. You can also display a value in a different number system if you change the output radix with CODE–R.

Changing the Output Radix

CODE-R changes the output radix or numerical base. All memory displays are in terms of current output radix, which is initially hexadecimal. CODE-R also causes the current value (=) to be displayed. To set the output radix to k, type:

k CODE-R

To reset the output radix to hexadecimal, type:

16. CODE-R

You can reset the output radix to hexadecimal by omitting the parameter to CODE-R. You can also type:

CODE-R

If the output radix is hexadecimal when CODE-R is typed with no parameters, then it is set to decimal.

The output radix only applies to numbers displayed by the Debugger; the interpretation of numeric constants you type in is independent of the output radix.

Examining and Modifying Registers

The processor register mnemonics name the current machine registers associated with the current process. For example, IP and FL name the instruction pointer and flags associated with the current process. You change PR to examine the registers of other processes by following the previously defined procedure.

Examine and change PR as any other word location: open it and enter a new parameter. For example, change the current process to process number 7, type:

PR ->4 7

The current process, originally 4, changes to number 7.

The Debugger interprets the words "next" and "previous" according to the mode in which a location is open. For example, next can mean next byte, next word, or next instruction.

Using Register Mnemonics

You can read and write to all of the registers listed below:

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

The following are general registers:

AX	Accumulator (has a high and low byte, AH and AL).
BX	Base (has a high and low byte, BH and BL).
CX	Count (has a high and low byte, CH and CL).
DX	Data (has a high and low byte, DH and DL).

The following are pointers and index registers:

SP	Stack Pointer
BP	Base Pointer
DI	Destination Index
SI	Source Index

The 16-bit FL register contains flags and the IP register contains the instruction pointer.

You can use register mnemonic symbols (AX, SI, and so on) on either side of an expression; however, the symbols must be the left-side values if you are making a change in a register.

Displaying Additional Memory

CODE-D provides additional memory display facilities. CODE-D takes two parameters and can display many bytes of memory, along with ASCII equivalents formatted into columns. To display *k* bytes of memory starting at location *addr*, type:

k, addr CODE-D

Reading and Writing To Ports

Use the left arrow and right arrow commands with port constants to read from and write to ports. For example, to read from the byte port 17i, type

17i ←

To read from the word input port 31i, type

31i →

In either case, the port becomes an open location and you can specify a new parameter that will 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 the character from the keyboard.

Displaying Linked-List Data Structures

The CODE-N command displays linked-list data structures. Use CODE-N with the internal Debugger registers CB and DB.

To display blocks of memory that are k bytes long and have a DS-relative link word at the jth byte, first set CB equal to k and DB to j. To display the first block, type:

addr CODE-N

To display each subsequent block, type:

CODE-N

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

n, addr CODE-N

Displaying a Trace of the Stack

The CODE-T command displays the procedure-invocation stack for the current process. To display the entire stack, type

CODE-T

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

k CODE-T

Displaying the Status of Processes and Exchanges

CODE-S gives the status of processes and exchanges. To display all of the processes and exchanges, type:

CODE-S

To see the list of processes or pointers to messages waiting on exchange k, type:

k CODE-S

When you list the processes, the following information appears:

id	The process identifier number.
oPcb	The address of the process control block for that process. The address relative to the operating system's data segment.
cs:ip	The address of the next instruction to be executed by the process.
link	A link address used by BTOS to keep processes threaded.
st	A byte containing status flags.
pr	The priority of the process.
ss:sp	The address of the top of the stack for the process.
exch	For the particular process, the default exchange for responses.

Forcing a Crashdumpfile

Occasionally it is necessary to get information from other parts of the system while you are debugging a process on a particular board. Since the Debugger only allows you to see one board at a time, you may need to force a remote board crash. This generates a crashdumpfile that will help you determine what other boards are doing while your process is executing. Setting CODE-Y and rebooting the system will produce the crashdumpfile, which records everything in memory at the point designated. If you use this, further debugging is possible without having to reboot the system. To set CODE-Y, enter:

slot number CODE-Y

You can force a crashdump on a local board; however, doing so will lock out the keyboard and you will have to reboot the system to continue debugging.

Setting and Processing Breakpoints

Breakpoints are addresses that the programmer sets within the code of a program to designate where to suspend a process. Generally, the user sets breakpoints in areas of program code which may be suspected of having execution problems. When these points are encountered during process execution, the process suspends and the Debugger is activated. Within the Debugger, the user can look at what is happening in memory at this address as this piece of code executes. A breakpoint stays in effect until it is explicitly removed.

If the debugger is operating in simple mode, all user processes are suspended whenever a breakpoint is encountered. If the Debugger is in multiple process mode, only the process that has encountered a breakpoint is suspended.

Setting and Querying Breakpoints

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

addr CODE-B

To obtain a display of the list of all of the breakpoints that are set at any given time, including CODE-I breakpoints, type

CODE-B

without an address parameter.

Clearing Breakpoints

To clear a breakpoint, type a parameter, followed by CODE-C. For example, to clear the breakpoint at address *addr*, type:

addr CODE-C

To clear all of the breakpoints at once, type

CODE-C

without an address parameter.

Proceeding From a Breakpoint

To proceed from the most recently found breakpoint in the current process, type

CODE-P

The breakpoint remains in place and the suspended 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 proceed, and to break the *k*th time the breakpoint is reached (instead of the next time it is reached), type

k CODE-P

where k is a decimal number.

Note: *CODE-P with no parameters is equivalent to CODE-P with a parameter of 1.*

To remove the breakpoint before proceeding, type

0 CODE-P

In simple mode, CODE-P causes the Debugger to become dormant until the next breakpoint is encountered or you reactivate it with either CODE-SHIFT key. Pressing the GO key is equivalent to pressing CODE-P.

In multiple process mode, CODE-P causes the Debugger to remain active; however, pressing GO causes the Debugger to become dormant.

Executing Instructions Individually

When you encounter a breakpoint, you can execute the next individual instruction in the current process and then break again by typing

CODE-X

After the system executes this instruction, it opens and displays the next instruction. From any breakpoint, you can type CODE-X repeatedly to see a series of instructions displayed and executed one by one; this is also known as single stepping through a process.

Use CODE-P to resume normal execution after using CODE-X.

Starting a Process at a Specified Address

To begin process execution at a different breakpoint at address *addr*, type

addr CODE-G

The address *addr* should be an expression that includes a user-defined public symbol (for example, "RgParam+5"), or have the form *x:y* where *x* is an appropriate CS parameter and *y* is an appropriate IP parameter. CS and IP are the address of the instruction currently executing.

The commands

10:0 CODE-G

and

100 CODE-G

both start execution at absolute address 100. However, the command "10:0 CODE-G" sets CS to 10, and the command "100 CODE-G" sets CS to 0.

In simple mode, CODE-G causes the Debugger to become dormant until the next breakpoint is encountered or you press CODE–SHIFT–A or CODE-SHIFT-B.

In multiple-process mode, CODE-G causes the Debugger to remain active. GO causes the Debugger to become dormant.

Setting Breakpoints In Interrupt Handlers

If you are debugging the operating system kernel or an interrupt handler, the previously discussed breakpoints are not sufficient to break processing. To set a breakpoint in the operating system kernel or in an interrupt handler at address addr, type

addr CODE-I

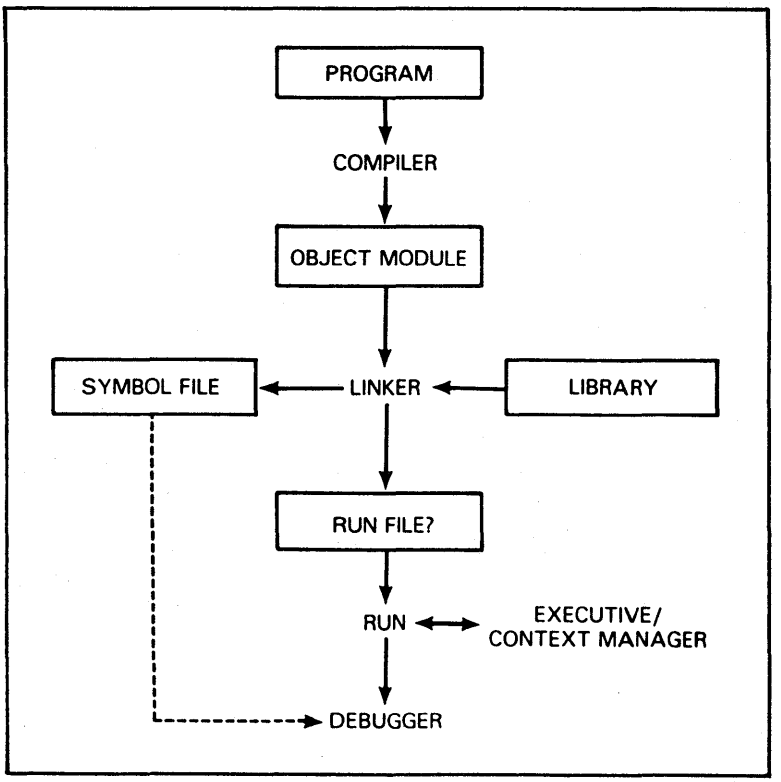
The standard keyboard and video facilities of the operating system support your interaction with the Debugger, except when the OS kernel or an interrupt handler is broken. At these breakpoints, all processes (including OS processes) are suspended, and the Debugger then works by direct access to the physical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

The Debugger uses hardware interrupts 1 and 3. Therefore, user programs should not use hardware interrupts 1 or 3. Other parts of the operating system use other hardware interrupts.

Debugger Relations to Other Program Steps

Figure 7-1 provides a diagram of the relationship between the Debugger and other steps in a program.

Figure 7-1 The Debugger in Relation to Other Steps in Executing a Program.



E7529

Command Summary

The Debugger parameters and commands and their effects are summarized in the following table.

Command (workstation)	Command (RS232-C)	Parameters	Effect
CODE-B	CONTROL-B		Displays a list of all breakpoints.
CODE-B	CONTROL-B	*addr*	Sets a breakpoint at addr when in simple or multi-process mode.
CODE-C	CONTROL-C		Clears all breakpoints.
CODE-C	CONTROL-C	*addr*	Clears the breakpoint at *addr*.
CODE-D	CONTROL-D	*k, addr*	Displays *k* bytes starting at *addr* CODE–E.
CODE-F	CONTROL-F		Turns symbolic output ON and OFF.
CODE-F CODE-F	CONTROL-F CONTROL-F	*'filename'* *paragraph, filename*	Uses the symbol file. Use symbol file with load offset.
CODE-F	CONTROL-F	*'filename'*	Uses the symbol file with load offset.
CODE-G	CONTROL-G	*addr*	Starts the current process at *addr*.
CODE-I	CONTROL-I		Displays a list of all breakpoints.
CODE-I	CONTROL-I	*addr*	Sets a breakpoint at *addr* in interrupt mode.

Command (workstation)	Command (RS232-C)	Parameters	Effect
CODE-K	CONTROL-K		Exits from and deactivates the Debugger.
CODE-N	CONTROL-N		Displays the next entry in a linked list.
CODE-N	CONTROL-N	*addr*	Displays the first entry in a linked list.
CODE-N	CONTROL-N	*k, addr*	Displays *k* entries in a linked list.
CODE-P	CONTROL-P		Proceeds from the current breakpoint.
CODE-P	CONTROL-P	*k*	Proceeds *k* times from the current breakpoint.
CODE-P	CONTROL-P	0	Proceeds after removing the current breakpoint.
CODE-R	CONTROL-R		Sets the output radix to hexadecimal (or to decimal, if the current radix is hexadecimal)
CODE-R	CONTROL-R	*k*	Sets output radix to *k*.
CODE-S	CONTROL-E		Displays status of all processes and exchanges.
CODE-S	CONTROL-S	*k*	Displays status of exchange *k*.
CODE-T	CONTROL-T		Displays a trace of the stack.
CODE-T	CONTROL-T	*k*	Displays a trace of *k* levels of the stack.

Command (workstation)	Command (RS232-C)	Parameters	Effect
CODE-X	CONTROL-X		Executes an instruction, and opens the next instruction.
left arrow	<	*addr*	Opens *addr* as a byte.
left arrow	<	*k, addr*	Opens *k* successive bytes.
right arrow	>	*addr*	Opens *addr* as a word.
right arrow	>	*k, addr*	Opens *k* successive words.
MARK	CONTROL-]	*addr*	Opens *addr* as an instruction.
MARK	CONTROL-]	*k, addr*	Opens *k* successive instructions.
-		value	Re-displays the current value.
up arrow	CONTROL-W		Opens the previous location.
up arrow	CONTROL-W	value	Changes the current location, and opens the previous location.
down arrow	CONTROL-V		Opens the next location.
down arrow	CONTROL-V	value	Changes the current location, and opens the next location.
GO	ESCAPE		Exit the Debugger and, if in simple mode, proceed from current breakpoint.
RETURN	RETURN		Close the open location.

Status Codes

Decimal Value	Meaning
1001	Cannot convert from simple mode to multiple process mode. To enter multiple process mode, first exit the Debugger and the press CODE-SHIFT-B.
1002	Not enough memory for multiple process mode or CODE-I breakpoint.
1003	Cannot deactivate the Debugger. You cannot deactivate the Debugger while CODE–I breakpoints are set or while a breakpoint has just executed. To deactivate the Debugger, first remove all CODE–I breakpoints and/or proceed (single step) from the breakpoint.
1004	Breakpoint already there. The Debugger allows only one breakpoint per location.

Quick Reference to Debugger Commands

B20	RS232-C	Effect
CODE-B	CONTROL-B	list all breakpoints
CODE-C	CONTROL-C	clears all breakpoints
CODE-D	CONTROL-D	displays *k* bytes
CODE-F	CONTROL-F	turn symbolic output on/off
CODE-G	CONTROL-G	with parameter, start current process at *addr*
CODE-I	CONTROL-I	set interrupt breakpoint
CODE-K	CONTROL-K	exit and deactivate Debugger, press twice
CODE-N	CONTROL-N	next entry in linked list
CODE-P	CONTROL-P	proceed from breakpoint
CODE-R	CONTROL-R	set output radix
CODE-S	CONTROL-E	display status
CODE-T	CONTROL-T	display trace of stack
CODE-X	CONTROL-X	execute and open next instruction
left arrow	<	open as bytes
right arrow	>	open as words
MARK	CONTROL-]	open as instruction
-		redisplay current value
up arrow	CONTROL-W	open previous location
down arrow	CONTROL-V	open next location
GO	ESCAPE	exit the Debugger
RETURN	RETURN	close open location

Glossary

Address expression. Description of a location in memory. The description consists of one or more symbols, or an indexed or nonindexed parameter.

Breakpoint. A user-defined point in the code for a process. Execution stops when a process reaches a breakpoint.

Byte pattern. User-defined group of byte specifiers. The specifiers are separated by commas and enclosed in double quotation marks.

Byte specifier. Sequence of two-digit hexadecimal numbers, or a string of characters enclosed in single quotation marks.

Clear. Remove a breakpoint from a particular location in memory.

Code listing. English-language display of code generated by a compiler or translator.

Crash dump. Output (memory dump) caused by a system failure.

Current process. The process identified by the PR register in the Debugger. Any registers that are read or written by the Debugger are for the current process.

Current value. The value most recently typed by the user, or the value most recently displayed by the Debugger.

Echo. Repetition on a line printer, in a file or on a screen of instructions entered by the user and/or material displayed by the Debugger.

Exchange. Path on which a process waits for or receives messages or communications from another process or processes.

Indexed address. Address expression that uses index registers.

Interrupt mode. Debugger operating mode used to debug interrupt handlers or to set breakpoints in the Operating System Kernel.

Link word. Word address, a 16-bit address) pointing to the next block of data.

Linked-list data structure. 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. Software system that loads and connects together the object programs output separately by a compiler or assembler, and from them produces a run file.

Multiple Process Mode. Debugger operating mode used in debugging a user task that involves more than one process and that depends on continuous execution of all processes except the ones stopped at breakpoints.

Offset. The number of bytes from the beginning of a segment to a specified memory location.

Output radix. The base of the notation in which Debugger output is expressed (binary, decimal, hexadecimal, or any other base from 2 to 16, inclusive).

Parameter. A constant (number, port, or text), a symbol, unary or binary operators, address expressions, or symbolic instructions.

Physical address. An address that does not specify a segment base, and is relative to memory location 0.

Pointer. See "segmented address".

Port constant. Number followed by an "i" or an "o" (indicating an input port and an output port, respectively).

Public symbol. An ASCII character string associated with a public variable, a public value, or a public procedure.

Public variable. Variable whose address can be referenced by a module other than the module in which the variable is defined.

Public value. Value whose address can be referenced by a module other than the module in which the value is defined.

Public procedure. Procedure whose address can be referenced by a module other than the module in which the procedure is defined.

Register mnemonic. Two-letter symbolic name for a register in the processor (for example, AX, BL, SI).

Run file. File created by the linker. The run file contains the initial image of code and data for a program.

Segment. A discrete portion of memory, of a routine, or of a program.

Segment address. Address of a segment base.

Segmented address (pointer). Address that specifies both a segment base and an offset.

Segment override. Operating code that causes the system to use the segment register specified by the prefix when executing an instruction, instead of the segment register that it would normally use.

Set. Place a breakpoint at a particular location in memory.

Simple mode. Debugger operating mode used in debugging a single-process user task, for example, debugging a Pascal run-file.

Stack. A region of memory, accessible from one end by means of a stack pointer.

Stack frame. 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 variables.

Stack pointer. The indicator to the top of a stack. The stack pointer is stored in the registers SS:SP.

Stack trace. Debugger display of a stack, organized by stack frames.

State variable. Symbolic name of a register that contains data indicating the state of the Debugger (for example, PR, IP, or FL).

Symbol. A series of alphanumeric and other characters (underscore, period, dollar sign, pound sign, or exclamation mark).

Symbolic instructions. Instructions containing symbols, which are mnenomic characters corresponding to assembly-language instructions. (These instructions cannot contain user-defined public symbols.)

System process. Any process that is not terminated when the user calls Exit.

Text constant. Sequence of characters enclosed by quotation marks.

User process. Any process that is terminated when the user calls Exit.

Index

A

B

C

C Cont.)

P

R

S

T

U

Title: _____

Form Number: _____ Date: _____

Burroughs Corporation is interested in your comments and suggestions regarding 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 _____

Title _____

Company _____

Address _____
Street City State Zip

Telephone Number ( ) _____
Area Code

Title: _____

Form Number: _____ Date: _____

Burroughs Corporation is interested in your comments and suggestions regarding 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 _____

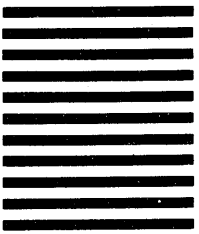
Title _____

Company _____

Address _____
Street City State Zip

Telephone Number ( ) _____
Area Code

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY CARD

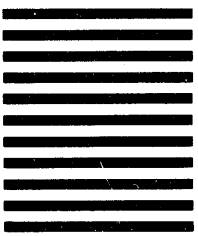
FIRST CLASS PERMIT NO. 817 DETROIT, MI 48232

POSTAGE WILL BE PAID BY ADDRESSEE

Burroughs Corporation
Production Services – East
209 W. Lancaster Avenue
Paoli, Pa 19301 USA

ATTN: Corporate Product Information

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY CARD

FIRST CLASS PERMIT NO. 817 DETROIT, MI 48232

POSTAGE WILL BE PAID BY ADDRESSEE

Burroughs Corporation
Production Services – East
209 W. Lancaster Avenue
Paoli, Pa 19301 USA

ATTN: Corporate Product Information