IBM

Series/1

SC34-0639-0

Event Driven Executive Problem Determination Guide

Version 5.0

Library Guide and Common Index SC34-0645	Installation and System Generation Guide SC34-0646	Operator Commands and Utilities Reference SC34-0644
Language Reference SC34-0643	Communications Guide SC34-0638	Messages and Codes SC34-0636
Operation Guide SC34-0642	Event Driven Language Programming Guide SC34-0637	Reference Cards SBOF-1625
Problem Determination Guide SC34-0639	Customization Guide SC34-0635	Internal Design LY34-0354

IBM

Series/1

SC34-0639-0

Event Driven Executive Problem Determination Guide

Version 5.0

Library Guide and Common Index SC34-0645	Installation and System Generation Guide SC34-0646	Operator Commands and Utilities Reference SC34-0644
Language Reference SC34-0643	Communications Guide SC34-0638	Messages and Codes SC34-0636
Operation Guide SC34-0642	Event Driven Language Programming Guide SC34-0637	Reference Cards SBOF-1625
Problem Determination Guide SC34-0639	Customization Guide SC34-0635	Internal Design LY34-0354

First Edition (December 1984)

Use this publication only for the purpose stated in the Preface.

Changes are made periodically to the information herein; any such changes will be reported in subsequent revisions or Technical Newsletters.

Summary of Changes for Version 5.0

The following additions and changes have been made to this document:

- A new section has been added to *Chapter 4, Analyzing and Isolating Run Loops*, which shows you how to examine an unmapped storage area in your program for the cause of a run loop.
- A new section has been added to *Chapter 6, Analyzing and Isolating a Program Check*, which shows you how to examine an unmapped storage area in your program for the cause of a system or application program check.
- *Chapter 7, Analyzing a Failure Using a Storage Dump* has been updated to include a description of how to interpret the unmapped storage information provided in the storage dump.
- *Chapter 9, Recording Device I/O Errors and Program Check Information* includes a description of how to record and print the contents of any program check messages that may occur on your system.
- A new appendix (Appendix B) has been added to the book which describes the hardware requirements and procedures for using the Remote Support Link feature of the Event Driven Executive. This feature enables an IBM support center representative to get direct access to your Series/1 system through a remote terminal.

About This Book

This book is a guide to assist you in determining the causes of problems you encounter while using the system. It explains how to use many of the diagnostic tools available to help identify the problem. Use this book when the *Messages and Codes* cannot point you to the source of the problem or the corrective action to take.

Audience

This book is intended for anyone who uses the Series/1 and encounters a hardware or software problem. The *Operation Guide* describes how you can recognize symptoms of the problems discussed in this book.

How This Book Is Organized

This book contains 9 chapters and 3 appendixes:

- *Chapter 1. Some Things You Should Know About Problem Determination* overviews the process of problem determination.
- *Chapter 2. Determining the Problem Type* presents some common problem symptoms that can help you determine the type of problem you encounter.
- *Chapter 3. Analyzing and Isolating an IPL Problem* describes some procedures that can help identify the cause of an IPL failure.

How This Book Is Organized *(continued)*

- *Chapter 4. Analyzing and Isolating Run Loops* explains how to pinpoint the cause of run loop in an application program.
- *Chapter 5. Analyzing and Isolating a Wait State* describes how to determine the cause of a wait state during normal system operation.
- *Chapter 6. Analyzing and Isolating a Program Check* discusses how to isolate the cause of a system or application program check.
- *Chapter 7. Analyzing a Failure Using a Storage Dump* describes how to to read a stand-alone or $TRAP storage dump to isolate failures.
- *Chapter 8. Tracing Exception Information* explains how you can isolate the cause of exceptions by analyzing the software trace table CIRCBUFF.
- *Chapter 9. Recording Device I/O Errors and Program Check Information* discusses the use of the $LOG utility to record device I/O errors and program check messages.
- *Appendix A. How to Use the Programmer Console* describes the functions of the optional Series/1 programmer console and how you can use it during problem analysis.
- *Appendix B. Allowing IBM Access to Your System* describes the hardware requirements and procedures for using the Remote Support Link feature of the Event Driven Executive. This feature enables an IBM support center representative to get direct access to your Series/1 system through a remote terminal.
- *Appendix C. Conversion Table* contains a conversion table for hexadecimal, binary, EBDCIC, and ASCII equivalents of decimal values.

Aids in Using This Book

Several aids are provided to assist you in using this book:

- A Glossary that defines terms and acronyms used in this book and in other EDX library publications.
- An Index of topics covered in this book.

A Guide to the Library

Refer to the *Library Guide and Common Index* for information on the design and structure of the Event Driven Executive library and for a bibliography of related publications.

Contacting IBM about Problems

You can inform IBM of any inaccuracies or problems you find with this book by completing and mailing the **Reader's Comment Form** provided in the back of the book.

If you have a problem with the Series/1 Event Driven Executive services, you should fill out an authorized program analysis report (APAR) as described in the *IBM Series/1 Software Service Guide*, GC34-0099.

Contents

Contents

Figures

Figures

Chapter 1. Some Things You Should Know About Problem Determination

Problem determination involves analyzing a software or hardware error. The system can indicate in various ways that a problem exists. The two most common ways are by displaying messages on a terminal or by returning a return code to your application program. By using the *Messages and Codes* manual *before* you use this book, you may be able to determine the type of problem you have and the corrective action to take. If, however, you cannot determine the type of problem you have or how to correct it, use this book.

This book can help you isolate the cause of an error and indicate what actions you need to take to correct the error.

The cause of an error may not always be immediately apparent. An error may occur in an IBM-supplied software component, a hardware unit, or in an application program. A software component refers to programs or program modules such as $EDXASM, $S1ASM, $EDXLINK, and the rest of the software you install on your Series/1. A hardware unit refers to a particular device attached to your Series/1. Application programs are programs you write.

Some problems you encounter may require you to place a service call. However, by using this book before you place a call for service:

- You might be able to correct the problem and continue operations.
- You might be able to circumvent the problem while you arrange for servicing.

Some Things You Should Know About Problem Determination

- You may find that the problem is caused by equipment or programming other than that supplied by IBM.
- The information you gather can reduce the time it takes to correct the problem if you do call for service.

EDX provides various aids, such as utilities and operator commands, that help you to pinpoint the source of a problem. The programmer console, an optional Series/1 hardware feature, enables you to perform more extensive analysis.

Some of the topics presented in this book show the use of the programmer console in analyzing problems. For more information on using this feature, see Appendix A, "How to Use the Programmer Console" on page PD-127.

To start the problem investigation, turn to Chapter 2, "Determining the Problem Type" on page PD-3.

Chapter 2. Determining the Problem Type

Before you begin analyzing a problem, you must determine the type of problem you have. Some problem types you encounter may be very apparent while others may not be so apparent. The following section presents some problem indicators and symptoms to help you determine the problem type.

Some Hints to Determine the Possible Problem Type

To help you determine your problem type, review the following problem indicators and symptoms. After reviewing these items and finding the indicator or symptom that best describes your problem, turn to the chapter indicated. The chapter you are referred to will help you to further analyze and isolate the problem.

Can You Operate the System After Pressing the Load Button?

When you press the Load button on your Series/1, the system performs an initial program load (IPL). When the IPL process ends, the system is ready for use. If you cannot use the system after attempting an IPL, see Chapter 3, "Analyzing and Isolating an IPL Problem" on page PD-5.

Determining the Problem Type

Some Hints to Determine the Possible Problem Type *(continued)*

Is the Run Light On and Solidly Lit?

When the Series/1 performs an operation, the Run light is on. Typically, the Run light flickers on and off during the operation. However, if you observe that the Run light remains on with a steady glow, the system or your program may be in a loop. If this is your problem symptom, Chapter 4, "Analyzing and Isolating Run Loops" on page PD-17 will help you isolate this problem type.

Is the System or a Program Idle While You Expect Activity?

When the Series/1 is not performing any operation or servicing an interrupt, the Wait light is on. The Wait light indicates the system is inactive. If, however, you notice the Wait light on solidly while programs should be active, the system or a program is probably in a wait state. Another symptom indicating a wait state is that you do not receive the "greater than" symbol (>) after you press the attention key on your terminal. If your system or program has these symptoms, see Chapter 5, "Analyzing and Isolating a Wait State" on page PD-33.

Did the System Issue a Program Check Message?

When the system encounters an abnormal condition, it issues a program check message. Two kinds of program checks can occur: a system program check or an application program check. The system displays the program check message on the $SYSLOG device. The system also records the program check message in a log data set if $LOG is active.

If you observe a program check message, Chapter 6, "Analyzing and Isolating a Program Check" on page PD-43 can help you isolate the problem.

Chapter 3. Analyzing and Isolating an IPL Problem

If your system fails to IPL correctly, there are a number of possible causes. This chapter presents some problem symptoms and procedures that can help you identify the failing area and provide help in solving the problem.

What You Should Check First

Before you begin troubleshooting the problem, review the items in the following list. By ensuring that these items are correct, you may be able to pinpoint the problem immediately:

- Is the power switch in the ON position for all devices?
- Is the IPL Source switch in the correct position for the device from which you are trying to IPL?
- For diskette IPL, is the IPLable diskette inserted correctly?
- For diskette IPL, is the door on the diskette device closed?
- If this is a new installation (EDX is not installed) and you are trying to IPL the starter system, verify with your service representative that the devices are at the addresses supported in the starter system. Refer to the Program Information Department (PID) directory or the *Installation and System Generation Guide* for the device addresses.

What You Should Check First *(continued)*

- If EDX is already installed and the supervisor *previously* IPLed, does a backup supervisor (or starter system) IPL from the alternate IPL device? If the alternate device IPLs, go to the section "How to Recognize a Problem with the IPL Device."

- If the starter system IPLs but your tailored supervisor does not IPL, go to the section "Determining the Failure in a Tailored Supervisor" on page PD-8.

If the previous items do not point out the problem, the problem may lie in the IPL device, IPL text, the supervisor, or other attached devices. The following sections describe how to isolate problems in these three areas.

How to Recognize a Problem with the IPL Device

If the Load light remains on and you cannot IPL from the primary and the alternate IPL device and you have ensured that all the items in the section "What You Should Check First" on page PD-5 are correct, call your service representative for corrective action. This symptom indicates that the hardware could not read the IPL text (bootstrap program) from the IPL device. If you have a programmer console, you may also notice that the console lights indicate either X'E0' or X'E5'. The value X'E0' indicates that there is a hardware problem with the IPL device. The value X'E5' may indicate either a hardware or software problem.

If you can IPL from one IPL device, the following procedures can help you determine if the failure is due to:

- No IPL text written when the disk or diskette was initialized

- Defective IPL text

- IPL text points to an invalid supervisor

- Hardware problem on that IPL device.

How to Recognize a Problem with the IPL Device *(continued)*

How to Correct the IPL Text

Use the following procedure to correct the IPL text:

1. Set the IPL Source switch for an IPL from the device from which you can IPL.

2. Press the Load button to IPL the system.

3. Load $INITDSK and rewrite the IPL text (II command) to the failing IPL device.

4. Set the IPL Source switch to IPL from the failing IPL device.

5. Press the Load button to IPL the system.

If this procedure does not correct the IPL problem, the problem may be with the supervisor on the failing IPL device or it may be a hardware problem. By reloading the supervisor, you may correct the problem. How to do this is described next.

How to Reload the Supervisor

Use the following procedure to reload the supervisor:

1. Set the IPL Source switch for an IPL from the device from which you can IPL.

2. Press the Load button to IPL the system.

3. Load $COPYUT1 and copy (CM command) the IPLable supervisor from the current IPL device to the failing IPL device. Copy also $LOADER and any initialization modules you require.

4. Load $INITDSK and rewrite the IPL text (II command) to point to the supervisor you copied to the failing IPL device.

5. Set the IPL Source switch to IPL from the failing IPL device.

6. Press the Load button to IPL the system.

If this procedure does not correct the IPL problem, you have a hardware problem with that IPL device. Call your service representative for corrective action.

Analyzing and Isolating an IPL Problem

Determining the Failure in a Tailored Supervisor

Review the following items before you begin analyzing the failure:

- Did you receive a -1 completion code (successful) from the system generation assembly and link-edit?
- Did you include all the modules you need (on the INCLUDE statements) to support the attached devices?
- Is $EDXNUC the first seven characters of the $XPSLINK output?
- Does this tailored supervisor fail to IPL, although it did IPL previously? If it did IPL previously, go to the section "How to Recognize a Problem with the IPL Device" on page PD-6.
- If this tailored supervisor never IPLed, the following sections may assist you in isolating the failure. In order to use this information, however, you must have a programmer console or be able to use the $D operator command (in partition 1) after the IPL failure.

If you do not have a programmer console but can use the $D operator command (in partition 1) after the IPL failure, go to the section "Analyzing the INITTASK Task Control Block" on page PD-11.

If you have a programmer console, begin with the section "Detecting an IPL Stop Code Error" on page PD-9.

If you do not have a programmer console and cannot use $D after the failure, do the following:

1. IPL the starter system.
2. Load $IOTEST and verify all hardware configured and their addresses (LD command).
3. Review the system generation listing and ensure that all devices are defined correctly and that all modules required to support those devices are included.

Detecting an IPL Stop Code Error

If the system encounters an error during terminal initialization or it encounters an error within the cross-partition supervisor you are trying to IPL, the error could cause the system to enter a run loop or a wait state. For example, the error could be caused by a defective attachment card or perhaps a missing random access memory load module. When such errors exist, the system issues a stop code. The stop code can help you identify which area is failing.

This section explains how to determine if the failure is due to a stop code error. You will need a programmer console to perform this step.

To determine if the IPL failed because of a stop code, follow these procedures:

1. Set the IPL Source switch to point to the device from which you will IPL.

2. Set the Mode switch to Diagnostic mode position.

3. If the IPL is from diskette, insert the IPL diskette and close the door on the diskette device.

4. Press the Load button.

   If the system encounters a stop code condition, the processor will stop. The Stop light also comes on.

5. Press the Op Reg button on the programmer console.

After pressing the Op Reg button, the stop code is displayed in the indicator lights. The stop code is in the form X'64nn'. The nn portion indicates the error condition. Refer to the *Messages and Codes* manual for an explanation of the stop code and the corrective action.

The next section presents another method you can use to determine if a terminal is the cause of the failure.

Determining the Failure in a Tailored Supervisor *(continued)*

Isolating a Failing Terminal Using the Terminal Control Block

This procedure enables you to determine if the system fails to initialize a terminal. The terminal control block (CCB) may point to the failing terminal. To help you detect if a terminal is causing the problem, you need the system generation link map listing for your supervisor. Look in the link map and find the address of the entry NEXTERM in module TERMINIT.

Using the programmer console, do the following:

1. Press the Reset key.

2. Press the Stop On Address key.

3. Enter the address of NEXTERM.

4. Press the Store key.

5. IPL the system. Each time the processor stops, the terminal whose terminal control block (CCB) address is in register 3 (R3) has been successfully initialized.

   If the processor does not stop, the failure occurred prior to terminal initialization. If this is the case, go to the section "Analyzing the INITTASK Task Control Block" on page PD-11.

6. When the processor stops, press R3 on the programmer console to determine which terminal was initialized. The address shown in R3 will match a CCB address in the section $EDXDEF of the link map. The name of the terminal also appears beside the address.

   If R3 does not contain a CCB address and you have overlay support, press Start. When the processor stops, press R3 again. Repeat this step until R3 contains a CCB address.

7. Press Start after checking off the CCB address in your link map. The system initializes each terminal in the order the terminals are specified in $EDXDEFS data set during system generation.

8. If the system then enters a run loop or a wait state, the terminal whose address follows the last CCB that you checked off is probably the cause of the problem.

   Ensure that all required initialization modules (if any) for that terminal were included during system generation. Also check to see if that terminal is defined correctly on the TERMINAL statement. If both the terminal and the support modules are defined correctly, call your service representative for corrective action on that terminal or attachment.

9. If the system does not enter a run loop, go to step 6 on page PD-10 .

If you still cannot identify the cause of the IPL failure using the previous procedure, go to the section "Analyzing the INITTASK Task Control Block."

Analyzing the INITTASK Task Control Block

The technique discussed in this section requires you to examine the INITTASK task control block. By examining this control block, you may be able to identify the cause of the IPL failure. INITTASK is the label of the task control block (TCB) used by the system initialization routines. The address of INITTASK (in module EDXSTART) is in the supervisor link map from system generation.

If you have a programmer console, begin with the section "Storing the Address of INITTASK" on page PD-12.

If, after the IPL failure has occurred, you can press the attention key, enter $D from a terminal in partition 1, and receive a prompt for input, go to the section "Displaying the INITTASK Task Control Block with $D."

Displaying the INITTASK Task Control Block with $D

Do the following when you receive the prompt ENTER ORIGIN: from $D:

1. Enter **0000**.

The next prompt, ADDRESS,COUNT:, asks you for an address and the number of words you want to display.

2. For ADDRESS, enter the address for INITTASK shown in the supervisor link map.

3. For COUNT, enter the value **14**. This value represents the first 14 words in the INITTASK TCB.

The system then displays the 14 words of information.

4. Record all the values displayed on the terminal.

5. Reply **N** to the prompt ANOTHER DISPLAY?

6. Go to the section "Interpreting the Task Control Block Information" on page PD-13.

Determining the Failure in a Tailored Supervisor *(continued)*

Storing the Address of INITTASK

After you locate the address of INITTASK in the supervisor link map, do the following at the programmer console:

1. Press the Stop key.
2. Press the AKR key.
3. Enter X'0'.
4. Press the Store key.
5. Press the SAR key.
6. Enter the address of INITTASK.
7. Press the Store key.

The next step is to display the contents of the INITTASK task control block.

Displaying the INITTASK Task Control Block using the Programmer Console

By displaying the values contained in the INITTASK task control block, you may get a clue as to what is causing the IPL failure.

The procedure discussed here requires you to display and record the first 14 words of information in the INITTASK TCB.

To read the first word of the TCB:

1. Press the Main Storage key. The contents is displayed in the indicator lights.
2. Record the value displayed in the indicator lights.

   Each time you press the Main Storage key, a new value is displayed.

3. Repeat the two previous steps 13 more times to obtain the remaining values in the TCB.

Determining the Failure in a Tailored Supervisor *(continued)*

Interpreting the Task Control Block Information

The first three words (words 0–2) of the INITTASK TCB make up the event control block (ECB). The next 11 words (words 3–13) contain the level status block (LSB) information. This 14-word area appears as follows:

Word 0–2	ECB
Word 3	IAR
Word 4	AKR
Word 5	LSR
Word 6	R0
Word 7	R1
Word 8	R2
Word 9	R3
Word 10	R4
Word 11	R5
Word 12	R6
Word 13	R7

The information in the LSB (words 3–13 of the TCB) is what you use to identify the failure. Since many of the system initialization modules are written in EDL, the register contents usually indicate the following:

IAR — The instruction address register (IAR) contains the address of the last machine instruction the system executed when the failure occurred.

AKR — The last 3-hexadecimal digits indicate in which address space operand 1, operand 2, and the IAR reside. Bit 0 of the AKR is the equate operand spaces (EOS) bit. If bit 0 is set to 1, the address space key indicated for operand 2 is the address space key used for operand 1 and operand 2.

LSR — The value of level status register (LSR). The bits, when set, indicate the following:

- Bits 0–4 — The status of arithmetic operations. Refer to the processor description manual for the meanings of these bits.
- Bit 8 — Program is in supervisor state.
- Bit 9 — Priority level is in process.
- Bit 10 — Class interrupt tracing is active.
- Bit 11 — Interrupt processing is allowed.

Bits 5–7 and bits 12–15 are not used and are always zero.

Determining the Failure in a Tailored Supervisor *(continued)*

R0 Because the supervisor uses this register as a work register, the contents are usually not significant.

R1 The address in storage of the last EDL instruction executed in the initialization module when the failure occurred.

R2 The address in storage of the active task control block (TCB).

R3 The address in storage of EDL operand 1 of the failing instruction.

R4 The address in storage of EDL operand 2 (if applicable) of the failing instruction.

R5 The EDL operation code of the failing instruction. The first byte contains flag bits which indicate how operands are coded. For example, the flag bits indicate whether the operand is in #1, #2, or is specified as a constant. The second byte is the operation code of the EDL instruction.

R6 Because the supervisor uses this register as a work register, the contents are usually not significant. However, you can determine if the system was emulating EDL code when the failure occurred if R6 is twice the value shown in the second byte of R5. For example, if the second byte of R5 contained X'32' *and* the system was emulating EDL, R6 would contain X'0064'.

R7 The supervisor uses this register as a work register. However, in many cases, R7 may contain the address of a branch and link instruction. The address may give you a clue as to which module passed control to the address in the IAR.

After you record all the TCB values, compare the value you recorded for R2 against the address of INITTASK. If these addresses do not match, you either have the wrong storage area or wrong link map.

If R2 does contain the address of INITTASK, start looking at the addresses in the remaining registers for a possible clue. Not all the registers may point to the failing area, but you should check the addresses that the registers point to nevertheless. Comparing the addresses you recorded and the addresses in the supervisor link map can help you identify the failure.

You can generally get an idea of which device is failing by the name or names of the supervisor modules. For example, if several of the addresses you recorded point to disk routines, you could assume that the IPL failure was related to a disk device.

The following discussion illustrates how the register contents can identify the problem area.

In this example, the IPL failure occurred because a disk device was defined incorrectly during system generation. The registers in the INITTASK TCB, and what they pointed to in the link map, are shown in Figure 1 . The registers that did not help identify the problem in this example are shown as "not applicable".

Register	Address	Module pointed to by register
IAR	X'27FA'	TAPE060 in DISKIO module
AKR	X'0000'	(not applicable)
LSR	X'88D0'	(not applicable)
R0	X'0000'	(not applicable)
R1	X'77BE'	DSKINIT1 in module DSKINIT2
R2	X'20DE'	INITTASK in module EDXSTART
R3	X'709A'	DINITDS1 in module DISKINIT
R4	X'06BA'	DMDDB in module $EDXDEF
R5	X'0000'	(not applicable)
R6	X'0000'	(not applicable)
R7	X'27F6'	TAPE060 in DISKIO module

Figure 1. Sample INITTASK Register Contents

Notice that the names of the supervisor modules are all disk related. Since the address in R4 (X'06BA') in this example is within the module $EDXDEF, you can identify exactly which device is causing the failure as follows:

1. Subtract the address of $EDXDEF from the address in R4. The link map showed that $EDXDEF is at address X'052E'. Thus, the resulting address is X'0188'.

2. Using the resulting address from step 1 and the assembly listing, look at the device definition statement at that address and identify which device is defined. The device defined on the definition statement is the cause of the IPL failure.

As was previously mentioned, the disk device was defined incorrectly. The disk was defined as a 4963-23. It *should* have been defined as a 4963-64.

Analyzing and Isolating an IPL Problem

Determining the Failure in a Tailored Supervisor *(continued)*

No IPL Completion Messages on $SYSLOG

If R5 contains the value X'0016', the supervisor has issued a DETACH for INITTASK and has completed the IPL process. (X'0016' is the EDL operation code for a DETACH.) However, if no IPL completion messages were displayed on $SYSLOG, $SYSLOG may be the possible cause of the problem.

Ensure that $SYSLOG is at the address you specified for $SYSLOG during system generation.

If R5 is not X'0016' and R6 does not contain X'002C', look at the remaining TCB values and see what supervisor modules they point to. The names of the modules may give you a clue as to which device is failing.

Chapter 4. Analyzing and Isolating Run Loops

A loop is a sequence of instructions that the system executes a repeated number of times. Often in application programs, you may have a need to intentionally code a loop to manipulate data and then exit the loop based on some exit condition you establish. Occasionally, a system or programming error can cause the system to execute a sequence of instructions endlessly. This type of loop is not intended and when it occurs, you must isolate the cause. To isolate the cause of the loop, however, you must be able to identify the program.

This chapter explains how you can identify which program is in a run loop when multiple programs are active. In addition, this chapter shows you how to isolate a run loop using $DEBUG. If you already know which program is in a run loop, refer to the section "Using $DEBUG to Isolate a Run Loop" on page PD-20.

It is possible for the system to enter a run loop if a device generates more interrupts than the system can handle. The section "How to Detect Loops Caused by Device Interrupts" on page PD-32 explains how you can determine if device interrupts are the cause of a system run loop.

When the error is such that it causes the system to enter a loop and you cannot issue any operator commands from a terminal, you should take a stand-alone or $TRAP dump. Chapter 7, "Analyzing a Failure Using a Storage Dump" on page PD-71 explains how to determine system failures of this sort. Refer to the *Operation Guide* for details on taking a stand-alone dump. The *Operator Commands and Utilities Reference* explains how to invoke $TRAP.

Analyzing and Isolating Run Loops

How to Identify a Program in a Run Loop

This section explains how to identify which program is in a run loop when multiple programs are active. Two methods are discussed: using the programmer console and using the $C operator command.

Using the Programmer Console to Identify a Looping Program

Several steps using the programmer console will require you to stop all activity on the system. Before you begin, consider what effect stopping the system will have on any active programs, in particular, any time-dependent programs.

To identify the looping program, do the following:

1. Press the attention key and enter the $A ALL operator command.

2. Write down the program names and their load point for each partition.

3. Set the Mode switch on the console to the Diagnostic position.

4. Look at the Level indicators for levels 0–3 on the programmer console. You may notice a particular level indicator showing more activity (pulsing more) than the other Level indicators. Further, you may notice a particular Level indicator pulsing at the same time the Run light is on. Noticing these indicators can help you determine on which hardware level the looping program is running.

   Note: Programs generally run on level 2 (the default) and level 3. Programs with an attention list task active (ATTNLIST instruction) run on level 1.

5. Press Stop on the programmer console. If the Level indicator light is on for the level on which you suspect the program is running (determined in step 4), go to step 6.

   If the Level indicator light is not on, continue pressing Start and Stop until the light is on, then go to step 6.

6. Press R1; a value is displayed.

How to Identify a Program in a Run Loop *(continued)*

7. Record the hexadecimal address displayed in the lights.

   To identify which program is at the address displayed for R1, you must determine the partition number:

   a. Press AKR.

   b. Press the Level indicator for the level you determined in step 4 on page PD-18 .

   c. Record the sum of the hexadecimal value displayed in lights 5–7. The number of the partition in which the program is running is 1 plus the value shown in lights 5–7. For example, if the sum of the lights had the value X'3', the partition number is partition 4.

8. Do steps 5 through 7 several times. This sequence will give you a range of instruction addresses. By comparing these addresses to the program load point addresses from step 1 on page PD-18, you can get an idea of which program might be looping and some of the instruction addresses within the loop.

After you have identified which program is in a run loop, you must determine where in the program the loop starts. The section "Using $DEBUG to Isolate a Run Loop" on page PD-20 explains how to do this.

Using $C to Identify a Looping Program

The purpose of using $C is to identify the looping program through a process of elimination.

Before you begin canceling programs, consider what impact that may have on any programs running normally. Also, consider whether you can recreate the environment from when the loop began. You may be able only to identify the failing program and not be able to analyze it until that program fails again. It is possible that the loop could be caused by this particular mix of running programs. When this is the case, canceling programs may make it harder to determine the cause of the loop. Consider taking a stand-alone or $TRAP dump as an alternative to $C.

When you issue $C, first cancel the programs you suspect are least likely to cause the problem. If the run loop condition still exists, continue canceling programs until the problem goes away. The last program you canceled is probably the cause of the run loop.

After canceling the program that caused the run loop, run that program again in an attempt to recreate the loop, then go to "Using $DEBUG to Isolate a Run Loop" on page PD-20.

If you cancel all but one program and the run loop condition still exists, go to the section "Using $DEBUG to Isolate a Run Loop" on page PD-20.

Analyzing and Isolating Run Loops

Using $DEBUG to Isolate a Run Loop

This section explains how to isolate a run loop with $DEBUG. The $DEBUG utility is described in detail in the *Operator Commands and Utilities Reference*. To show some techniques of isolating a run loop with $DEBUG, a sample program, MYPROG, is presented. The sample program contains a coding error which causes it to loop.

The sample program should display a prompt message requesting up to 40 characters of input data. After receiving input, the program should insert a blank between each character and then display the data. You end the program by entering a /*.

You will need the compiler listing for your program when using $DEBUG. Figure 2 shows the compiler listing for the sample program MYPROG.

The first step in isolating a run loop is to determine the starting point and ending point of the instructions causing the loop. How you do this using $DEBUG is discussed in the section "Determining the Starting and Ending Points of the Loop" on page PD-21.

```
LOC     +0   +2   +4   +6   +8
                                          PRINT      NODATA
0000    0008 D7D9 D6C7 D9C1 D440  MYPROG  PROGRAM    LABEL1
0034                              LABEL1  EQU        *
0034    8026 1A1A C5D5 E3C5 D940          PRINTEXT   'ENTER UP TO 40 CHARACTERS@'
0052    8026 1C1C C5D5 E3C5 D940          PRINTEXT   'ENTER A ''/*'' TO END PROGRAM@'
0072                              LABEL2  EQU        *
0072    402F 00D6 0000                    READTEXT   INPUT,PROMPT=COND
0078    A0A2 00D6 615C 00D0               IF         (INPUT,EQ,C'/*'),GOTO,LABEL4
0080    005A 0151 00D5                    MOVE       COUNT+1,INPUT-1,(1,BYTE)
0086    835C 0000 00D6                    MOVEA      #1,INPUT
008C    835C 0002 0100                    MOVEA      #2,OUTPUT
0092                              LABEL3  EQU        *
0092    065A 0000 0000                    MOVE       (0,#2),(0,#1),(1,BYTE)
0098    8332 0002 0001                    ADD        #2,1
009E    025A 0000 0152                    MOVE       (0,#2),BLANK,(1,BYTE)
00A4    8332 0000 0001                    ADD        #1,1
00AA    8332 0002 0001                    ADD        #2,1
00B0    A0A2 0150 0000 00C2               IF         (COUNT,NE,0),THEN
00B8    8035 0150 0001                      SUB      COUNT,1
00BE    00A0 0092                           GOTO     LABEL3
                                          ENDIF
00C2    0026 0100                         PRINTEXT   OUTPUT
00C6    902A 0001 0000                    PRINTEXT   SKIP=1
00CC    00A0 0072                         GOTO       LABEL2
00D0                              LABEL4  EQU        *
00D0    0022 FFFF                         PROGSTOP
00D4    2828 4040 4040 4040 4040  INPUT   TEXT       LENGTH=40
00FE    5050 4040 4040 4040 4040  OUTPUT  TEXT       LENGTH=80
0150    0000                      COUNT   DATA       F'0'
0152    40                        BLANK   DATA       C' '
0154    0000 0000 0000 0234 0000          ENDPROG
                                          END
```

Figure 2. Sample Program Compiler Listing

Using $DEBUG to Isolate a Run Loop *(continued)*

Determining the Starting and Ending Points of the Loop

While the program is running and in a loop, do the following:

1. Load $DEBUG in any available partition.

Try to load $DEBUG from a terminal other than the terminal from which the looping program was loaded. If you cannot use a different terminal, then load $DEBUG from the terminal used by the looping program.

2. Enter the name of the looping program when $DEBUG asks you for a program name and volume. Because the program is already loaded, you do *not* need to enter the volume name.

3. When $DEBUG asks for a partition, enter the number of the partition which contains the looping program. If $DEBUG and the looping program are in the same partition, press the enter key.

4. Reply **N** when asked if you want a new copy of the program loaded.

The following example shows what you would enter if you loaded $DEBUG in partition 2, with the sample program MYPROG running in partition 1:

```
> $L $DEBUG
LOADING $DEBUG     31P,00:00:00, LP=B600, PART=2
PROGRAM (NAME,VOLUME): MYPROG
PARTITION (DEFAULT IS CURRENT PARTITION): 1
ALREADY ACTIVE AT B400
DO YOU WANT A NEW COPY TO BE LOADED? N
```

5. Press the attention key and enter **AT** to set the first breakpoint at the address of the program's entry point. The entry point is the address of the first operand of the PROGRAM statement. Enter **TASK** when you are prompted for an option.

The entry point for the sample program MYPROG is at address X'0034'. This sequence follows:

```
> AT
OPTION (*/ADDR/TASK/ALL): TASK
LOW ADDRESS: 34
```

6. Set the next breakpoint at the address of the last executable instruction. This will ensure that all instructions within the loop are traced by $DEBUG.

The last executable instruction for MYPROG is the PROGSTOP at address X'00D0'.

Using $DEBUG to Isolate a Run Loop *(continued)*

Because only the starting and ending points of the loop are needed at this point, the NOLIST and NOSTOP options are selected:

```
HIGH ADDRESS: DO
LIST/NOLIST: NOLIST
STOP/NOSTOP: NOSTOP
     1 BREAKPOINT(S) SET
```

7. Press the attention key and enter **GO**. $DEBUG displays the addresses of the instructions that the program executes.

An example showing the output that $DEBUG displays while tracing the sample program MYPROG follows. Notice that the low address (starting point of the loop) is X'0072'. The high address (ending point of the loop) is X'00CC'.

```
                •
                •
                •
TASK0154 CHECKED AT  0072          (low address)
TASK0154 CHECKED AT  0078
TASK0154 CHECKED AT  0080
TASK0154 CHECKED AT  0086
TASK0154 CHECKED AT  008C
TASK0154 CHECKED AT  0092
TASK0154 CHECKED AT  0098
TASK0154 CHECKED AT  009E
TASK0154 CHECKED AT  00A4
TASK0154 CHECKED AT  00AA
TASK0154 CHECKED AT  00B0
TASK0154 CHECKED AT  00C2
TASK0154 CHECKED AT  00C6
TASK0154 CHECKED AT  00CC          (high address)
TASK0154 CHECKED AT  0072
TASK0154 CHECKED AT  0078
                •
                •
                •
```

Figure 3. Sample Trace Addresses from $DEBUG

8. Ensure that *all* addresses displayed by $DEBUG are repeated at least once before you end $DEBUG. You end $DEBUG by pressing the attention key and entering **END**. When all the addresses have been repeated, you now have all the instructions within the loop.

9. Using the trace addresses from $DEBUG, try to determine the cause of the loop from the compiler listing. "Using the Compiler Listing to Locate the Loop" on page PD-23 explains how you use the trace addresses to follow the logic of the loop.

The section "Some Common Causes of Run Loops" on page PD-23 gives some hints as to what might be the cause of the loop.

Some Common Causes of Run Loops

Run loops are often caused by some exit condition not being met within a program. The reason the exit condition is not met could be any of the following:

- Counters or variables that are never initialized when the program begins.
- Counters or variables that are not tested for an exit condition.
- Counters that never reach the limit you expected.
- Control passed to the wrong label in the program.

Check your program listing to be sure that none of the previous logic errors exist. If you cannot immediately pinpoint any of these conditions, continue reading this chapter.

Using the Compiler Listing to Locate the Loop

The compiler listing and the trace addresses displayed by $DEBUG enable you to follow the flow of the loop. Do the following steps to determine the problem:

1. Locate in the compiler listing, the lowest trace address displayed by $DEBUG. The lowest address for the sample program, MYPROG, is X'0072' (see Figure 3 on page PD-22).

At address X'0072', the instruction executed is a READTEXT.

```
LOC     +0    +2    +4    +6    +8
                                                 •
                                                 •
                                                 •
0034    8026  1A1A  C5D5  E3C5  D940             PRINTEXT  'ENTER UP TO 40 CHARACTERS@'
0052    8026  1C1C  C5D5  E3C5  D940             PRINTEXT  'ENTER A ''/*'' TO END PROGRAM@'
0072                                     LABEL2  EQU       *
0072    402F  00D6  0000                         READTEXT  INPUT,PROMPT=COND
0078    A0A2  00D6  615C  00D0                   IF        (INPUT,EQ,C'/*'),GOTO,LABEL4
                                                 •
                                                 •
                                                 •
```

The symptoms of the loop appear to be that the READTEXT did not allow you to enter input data when the program issued a message to do so.

Using the Compiler Listing to Locate the Loop *(continued)*

2. Again, reload $DEBUG in any available partition to determine the problem.

In this example, $DEBUG is loaded in partition 1, the same partition as MYPROG:

```
> $L $DEBUG
LOADING $DEBUG     31P,00:00:00, LP=B600, PART=1
PROGRAM (NAME,VOLUME): MYPROG
PARTITION (DEFAULT IS CURRENT PARTITION):
ALREADY ACTIVE AT B400
DO YOU WANT A NEW COPY TO BE LOADED? N
```

3. Press the attention key to set a breakpoint at the address following the READTEXT (address X'0078'):

```
> AT
OPTION (*/ADDR/TASK/ALL): ADDR
BREAKPOINT ADDR: 78
LIST/NOLIST: NOLIST
STOP/NOSTOP: STOP
      1 BREAKPOINT(S) SET
```

When the following message is displayed, $DEBUG has suspended the program's execution:

```
TASK0154 STOPPED AT  0078
```

At this point, you can look at any area of storage the program uses. If you set counters or variables in programs you run, examine those fields first. For MYPROG, you want to look at the number of characters the program read in as a result of the READTEXT.

The area labeled INPUT receives the input data upon a READTEXT:

```
LOC    +0   +2   +4   +6   +8
                                                •
                                                •
                                                •
0072   402F 00D6 0000                           READTEXT  INPUT,PROMPT=COND
                                                •
                                                •
                                                •
00D4   2828 4040 4040 4040 4040    INPUT        TEXT      LENGTH=40
```

4. Press the attention key and enter the following to see the number of characters stored in INPUT:

```
> LIST
OPTION (*/ADDR/R0...R7/#1/#2/IAR/TCODE/UNMAP): ADDR
ADDRESS: D4
LENGTH: 1
MODE(X/F/D/A/C): X
```

$DEBUG displays the following information:

```
00D4 X' 2800'
```

This information shows the length and count bytes for INPUT. The X'28' indicates the buffer size is 40 characters in length. However, the X'00' indicates that no characters were read in as a result of the READTEXT. If INPUT contained any data, the count byte would indicate the number of bytes.

Because INPUT contains no data, the problem might be either the TEXT statement coded for INPUT or the READTEXT instruction. Because you use READTEXT instructions to receive input data, the problem is probably with the READTEXT.

5. Review the description of READTEXT in the *Language Reference* to determine if the READTEXT is coded correctly. The READTEXT is coded as follows in the sample program:

```
READTEXT   INPUT,PROMPT=COND
```

The description for PROMPT=COND explains that when you use this operand, you must also code message text. No message text is coded on READTEXT in the sample program. The description further explains that when no message text is specified, READTEXT sets the count byte to zero and does not wait for input.

The sample program entered a run loop because the READTEXT is coded incorrectly. Isolating the run loop for this sample program is now complete.

6. Press the attention and enter **END** to end $DEBUG.

7. Cancel the looping program using the $C operator command.

8. Correct the coding error on the READTEXT as follows:

```
READTEXT  INPUT,'ENTER NEW DATA: ',PROMPT=COND
```

9. Recompile the program.

The techniques discussed up to this point in the chapter were useful in isolating the run loop in the sample program, MYPROG. The error, in this case, was somewhat obvious. However, you can apply these same techniques when the cause of a run loop in your program is not so apparent. The next section introduces additional techniques that may be helpful if you are trying to locate the cause of a run loop in a program that uses unmapped storage.

Examining an Unmapped Storage Area for the Cause of a Loop

A program may occasionally receive invalid or incorrect data. If the program is not prepared to handle such a situation, it could go into a run loop.

By using the LIST command of $DEBUG, you can examine the data areas in your program to see if any of the data in these areas is invalid or incorrect. (For more information on using the LIST command of $DEBUG, refer to the *Operator Commands and Utilities Reference*.) If the failing program uses unmapped storage, you may also want to look at the data in the unmapped storage areas. This section explains how to examine an unmapped storage area to find the cause of a run loop.

The sample program used in this section is called ADDNAMES. ADDNAMES processes a list of names and addresses which it reads from a data set into unmapped storage. The program should end when it encounters a -1 (X'FFFF') or when it processes more than 1,000 bytes of data. When ADDNAMES was loaded last, however, it went into a run loop. Figure 4 on page PD-27 shows the compiler listing for the sample program.

```
LOC     +0   +2   +4   +6   +8

0000    0008 D7D9 D6C7 D9C1 D440  ADDNAMES PROGRAM   START, DS=((DATA,DONORS))
000A    0000 0104 0184 0000 0000
0014    0188 0000 0001 0000 0100
001E    0186 0000 0000 0000 0000
0028    0000 0000 0000 0000 0000
0032    FFFF 0000 0000 0808 C4C1
003C    E3C1 4040 4040 0606 C4D6
0046    D5D6 D9E2 4040 0000 0000
0050    0000 0001 0000 0001 0000
005A    0000 0000 0000 0000 0000
006E    0000 0000 0000 0000
0076    0000 C1C1 0000 0000 0008  STORBLK1 STORBLK   TWOKBLK1=1,MAX=2
0080    0001 FFFF 0000 0000 0090
008A    0000 0000 0000 FFFF FFFF
0094    0000                      TOTAL    DC        F'0'
0096    0000                      LENGTH   DC        F'0'
                                  START    EQU       *
0098    00B9 0076 0000 0000 0101           GETSTG    STORBLK1,TYPE=ALL
00A2    035C 0000 0082                     MOVE      #1,STORBLK1+$STORMAP
00A8    80B9 0076 0001 0000 0300           SWAP      STORBLK1,1
00B2    8120 0000 0008 0000 020C           READ      DS1,(0,#1),8
00BC    0032
00BE    00A0 00CA 90A2 0094 03E8           DO        UNTIL,(TOTAL,GT,1000)
00C8    00F6
00CA    045C 0096 0000                       MOVE    LENGTH,(0,#1)
00D0    A0A2 0096 FFFF 00F6                  IF      (LENGTH,EQ,-1),GOTO,QUIT
00D8    E0A2 0096 0000 00F2                  IF      (LENGTH,GT,0),
                                           •
                                           •
                                           •
0100    8332 0000 0002                         ADD       #1,2
0106    0332 0000 0096                         ADD       #1,LENGTH
010C    0032 0094 0096                         ADD       TOTAL,LENGTH
0112                                         ENDIF
0112    00A0 00C2                          ENDDO
                                  QUIT     EQU       *
0116    00B9 0076 0000 0000 0201           FREESTG   STORBLK1,TYPE=ALL
0120    0022 FFFF                          PROGSTOP
                                           COPY      STOREQU
                                           •
                                           •
                                           •
```

Figure 4. Sample Program Compiler Listing

When $DEBUG is used to trace the execution of the program, the starting point of the loop (low address) is at X'00BE'. The ending point of the loop (high address) is at X'0112'. (The procedure for locating a run loop in a program is shown under "Determining the Starting and Ending Points of the Loop" on page PD-21.)

Examining an Unmapped Storage Area for the Cause of a Loop *(continued)*

The compiler listing for the sample program shows a DO instruction at address X'00BE'. The DO instruction marks the beginning of the loop. The loop ends with the ENDDO instruction at address X'0112'.

Looking at the contents of the DO loop, the program should be able to exit the loop when one of two conditions are met:

(1) The total length of the data read into storage exceeds 1000 bytes. At this point, the DO instruction at X'00BE' would satisfy the condition that it execute **until** the value in TOTAL is greater than 1000.

(2) The program finds a -1 in the data area. In this case, the IF instruction at X'00D0' would detect the condition and send the program to the label QUIT.

Since neither of these conditions occurred, it appears that the program had less than 1000 bytes of data to process but did not encounter a -1 when the data ended. Looking at the data in the unmapped storage area should reveal the source of the problem. To look at the contents of an unmapped storage area, do the following:

1. While the program is running and in the loop, load $DEBUG in any available partition.

   Try to load $DEBUG from a terminal other than the terminal from which the looping program was loaded. If you cannot use a different terminal, then load $DEBUG from the terminal used by the looping program.

2. Enter the name of the looping program when $DEBUG asks you for a program name and volume. Because the program is already loaded, you do *not* need to enter the volume name.

3. When $DEBUG asks for a partition, enter the number of the partition which contains the looping program. If $DEBUG and the looping program are in the same partition, press the enter key.

4. Reply N when asked if you want a new copy of the program loaded.

The sample program ADDNAMES is running in partition 1. In the following example, $DEBUG also is loaded in partition 1:

```
> $L $DEBUG
LOADING $DEBUG     45P,00:07:05, LP=B700, PART=1
PROGRAM (NAME,VOLUME): ADDNAMES
PARTITION (DEFAULT IS CURRENT PARTITION):
ALREADY ACTIVE AT 5C00
DO YOU WANT A NEW COPY TO BE LOADED? N
```

5. Press the attention key and enter **AT** to set a breakpoint at the address following the instruction that reads the data into unmapped storage.

   Note: Your program may be using several unmapped storage areas. If the SWAP instruction refers to a variable to find out the number of the unmapped storage area it should gain access to, check the contents of this variable to see which area was in use when the loop began.

In the sample program, the address of the instruction following the READ instruction is X'00BE':

```
> AT
OPTION (*/ADDR/TASK/ALL): ADDR
BREAKPOINT ADDR: BE
LIST/NOLIST: NOLIST
STOP/NOSTOP: STOP
      1 BREAKPOINT(S) SET
```

6. Press the attention key and enter **GO**.

$DEBUG displays a message when it suspends the program's execution at the breakpoint:

```
TASK0124 STOPPED AT  00BE
```

7. Press the attention key and enter the **LIST** command. After you enter this command, do the following:

   a. For "OPTION", enter **UNMAP**.

   b. For "STORBLK ADDRESS", enter the address of the STORBLK statement that defines the unmapped storage area you want to see.

   c. For "SWAP#", enter the number of the unmapped storage area you want to see.

   d. For "DISPLACEMENT", indicate how far from the beginning of the unmapped storage area the utility should go before listing the contents of the area. Enter a number of bytes (in hexadecimal). For example, if you enter **1A**, $DEBUG will begin the listing after the 26th byte in the unmapped storage area.

   e. For "LENGTH", enter the number of words, doublewords, or characters you want to list, depending on the MODE you select. Enter a decimal number.

   f. For "MODE", enter the format you want the data to appear in.

Examining an Unmapped Storage Area for the Cause of a Loop *(continued)*

The sample program reads eight 256-byte records into unmapped storage. The following example lists the first 256-byte record in the unmapped storage area:

```
> LIST
OPTION (*/ADDR/R0...R7/#1/#2/IAR/TCODE/UNMAP): UNMAP
STORBLK ADDRESS (0 TO CANCEL LIST): 76
SWAP#: 1
DISPLACEMENT: 0
LENGTH: 128
MODE(X/F/D/A/C): X
```

Figure 5 shows how $DEBUG displays the first record of the unmapped storage area for the ADDNAMES program.

```
0000 X' 0024 C54B C6C5 C4C5 D96B F1F2 F340 D5D6'
0010 X' D9E3 C840 C8C9 C3D2 D6D9 E86B D7D6 D9E3'
0020 X' E2D4 D6E4 E3C8 0026 D14B D7C9 E9E9 D6D3'
0030 X' C1E3 D66B F2F2 40E2 E8C3 C1D4 D6D9 C540'
0040 X' C4D9 4B6B C2C5 C1E5 C5D9 D2C9 D3D3 0020'
0050 X' D34B D3C9 D5C7 6B40 F5F5 F540 D4C1 C9D5'
0060 X' 40E2 E34B 6BC1 D3D3 C5D5 E3D6 E6D5 C540'
0070 X' 001E C44B C2C1 D2C5 D96B F1F2 40D5 D6D9'
0080 X' E3C8 40E4 D5C9 D6D5 6BD9 C5C4 D5C5 C3D2'
0090 X' 0028 D14B C1D5 E9C1 D3D6 D5C5 6BF5 F2F3'
00A0 X' 40E6 C5E2 E3E5 C9C5 E640 C2D3 E5C4 4B6B'
00B0 X' D9D6 C3D2 C3D9 C5C5 D26B 0000 0000 0000'
00C0 X' 0000 0000 0000 0000 0000 0000 0000 0000'
00D0 X' 0000 0000 0000 0000 0000 0000 0000 0000'
00E0 X' 0000 0000 0000 0000 0000 0000 0000 0000'
00F0 X' 0000 0000 0000 0000 0000 0000 0000 0000'
```

Figure 5. Sample Listing from $DEBUG

Each "logical record" that ADDNAMES processes consists of a name and address preceded by a "length" word. The length word indicates the length of the name and address in bytes. The program checks the length word, processes the amount of data that follows it, and moves to the next length word. The following is what the contents of the first logical record in Figure 5 would look like if they were translated into EBCDIC.

```
                 Data
E.FEDER,123 NORTH HICKORY,PORTSMOUTH    (Length - X'24' bytes)
```

If you were to list out the rest of the contents of the unmapped storage area, you would see that no more data exists. A brief examination of the storage contents in Figure 5 reveals that less than 1000 bytes of data were processed by the program. However, when you look for the second exit condition, a -1 (X'FFFF') at the end of the data, no -1 exists.

In the compiler listing for the ADDNAMES program, the first IF instruction in the DO loop looks for a -1 and the second IF instruction checks to see if the length of the data being processed is greater than 0. (See Figure 4 on page PD-27.) If no -1 is found, and if the length word contains only zeros, the program begins the loop again. Without a -1 to indicate the end of the data, the program preforms the DO loop endlessly.

In this case, the sample program obviously needs to be modified. However, to ensure that you have diagnosed the cause of the error correctly, you could place a -1 at the end of the data with the PATCH command of $DEBUG.

To use the PATCH command:

1. Press the attention key and enter **PATCH**.

2. After you enter the command, do the following:

   a. For "OPTION", enter **UNMAP**.

   b. For "STORBLK ADDRESS", enter the address of the STORBLK statement that defines the unmapped storage area you want to modify.

   c. For "SWAP#", enter the number of the unmapped storage area you want to modify.

   d. For "DISPLACEMENT", indicate how far from the beginning of the unmapped storage area the utility should go before listing the contents of the area. Enter a number of bytes (in hexadecimal). For example, if you enter **1A**, $DEBUG will begin the listing after the 26th byte in the unmapped storage area.

   e. For "LENGTH", enter the number of bytes, up to 16, that you want to modify. You cannot modify more than 16 bytes of data at a time. Enter a decimal number.

   f. For "MODE", enter the format you want the data to appear in.

3. The PATCH command displays the data to be modified. Enter your new data following the "DATA:" prompt message. Separate each word of data with a space.

   If you enter less data than the amount displayed, the command pads the remaining area with blanks (for character data) or zeros (for all other types of data).

Examining an Unmapped Storage Area for the Cause of a Loop *(continued)*

4. The command displays the data you entered and issues the prompt message "YES/NO/CONTINUE." Respond **Y** to confirm the change, **N** to cancel the change, or **CONTINUE** to confirm the change and to continue modifying data.

The following example uses the PATCH command to place a -1 at the end of the data in the unmapped storage area. After the change is made, program execution is resumed by pressing the attention key and entering **GO**.

```
> PATCH
OPTION (*/ADDR/RO...R7/#1/#2/IAR/TCODE/UNMAP): UNMAP
STORBLK ADDRESS (0 TO CANCEL PATCH): 76
SWAP#: 1
DISPLACEMENT: BA
LENGTH: 1
MODE(X/F/D/A/C): X
NOW IS
 00BA X' 0000'
DATA: FFFF
NEW DATA
 00BA X' FFFF'
YES/NO/CONTINUE: Y

> GO
```

How to Detect Loops Caused by Device Interrupts

The system can go into a run loop when device interrupts fill up the buffer area the system uses to contain interrupts. When this is the case, the loop begins at entry point SVCIBFOF in the supervisor module EDXSVCX.

If you have a programmer console installed, you can detect this condition by setting the Mode switch in the Diagnostic position while the system is looping. If the interrupt buffer becomes full, the system will stop and display a X'64FB' in the console indicator lights.

This run loop condition can be caused for two reasons:

1. The value you specified on the IABUF= operand of the SYSTEM statement (in $EDXDEFS) is not large enough to contain the number of interrupts. The default for IABUF= is 20. You may have to increase the value specified. Refer to the *Installation and System Generation Guide* for details on this operand.

2. A hardware problem on a device causes the device to send excessive interrupts which in turn causes IABUF to become full. Loading the $LOG utility, which records I/O errors, may identify the device experiencing errors. The $LOG utility is discussed in Chapter 9, "Recording Device I/O Errors and Program Check Information" on page PD-117.

Chapter 5. Analyzing and Isolating a Wait State

A wait state is a condition where the system or a program is waiting for the completion of an event or operation, but because of an error, the completion of the event or operation never occurs. When this condition exists, you must determine what prevented the event or operation from completing.

This chapter describes how to determine the cause of a wait state in an application program.

If, during a wait state, you press the attention key and the system does **not** display a "greater than" symbol (>), you should take a stand-alone or $TRAP dump. Chapter 7, "Analyzing a Failure Using a Storage Dump" on page PD-71 explains how you can determine the cause of the problem from the dump. Refer to the *Operation Guide* for details on taking a stand-alone dump. The *Operator Commands and Utilities Reference* explains how to invoke $TRAP.

In order to determine what caused the wait state in the application program, you must first find the address of the waiting instruction. How to do this is described next.

Analyzing and Isolating a Wait State

How to Find the Address of the Waiting Instruction Using $DEBUG

To find the address of the waiting instruction, do the following while the program is in the wait state:

1. Load $DEBUG in any available partition.

   Try to load $DEBUG from a terminal other than the terminal from which the waiting program was loaded. If you cannot use a different terminal, then load $DEBUG from the terminal used by the waiting program.

2. Enter the name of the waiting program when $DEBUG asks you for a program name and volume. Because the program is already loaded, you do *not* need to enter the volume name.

3. When $DEBUG asks for a partition, enter the number of the partition which contains the waiting program. If $DEBUG and the waiting program are in the same partition, press the enter key.

4. Reply **N** when asked if you want a new copy of the program loaded.

The following example shows what you would enter if the name of the program were WAITPGM and it was loaded in partition 1. $DEBUG, in this example, is loaded in partition 2:

```
> $L $DEBUG
LOADING $DEBUG      31P,00:00:00, LP=B600, PART=2
PROGRAM (NAME,VOLUME): WAITPGM
PARTITION (DEFAULT IS CURRENT PARTITION): 1
ALREADY ACTIVE AT B400
DO YOU WANT A NEW COPY TO BE LOADED? N
```

5. Press the attention key and enter the **WHERE** command. $DEBUG then displays the instruction address where the program is waiting. The following is an example of this sequence:

```
> WHERE
TASK1234 AT  00B8
```

6. Using the address displayed by $DEBUG, look at the compiler listing of that program to see what instruction is at that address.

7. Press the attention key and enter **END** to end $DEBUG.

After you identify the instruction that caused the wait, you must determine the reason why it was waiting. The following section can help you analyze the instruction that caused the wait state.

Analyzing the Instruction that Caused the Wait State

This section discusses how you can analyze the wait state if the program is stopped at any of the following instructions:

- ENQ
- ENQT
- WAIT.

If the program is not waiting on any of these instructions, go to the section "Other Possible Causes of a Wait State" on page PD-42.

Analyzing an ENQ Instruction

When the program is pointing to an ENQ instruction, you must examine the queue control block (QCB) the program tried to enqueue. By examining the queue control block, you can determine which task has control of that queue control block.

This section explains how to examine the queue control block when :

- The queue control block is defined within the program with a QCB statement.
- The queue control block is defined in the system common area, $SYSCOM.

Examining a Queue Control Block Defined in the Program

Do the following steps to examine the queue control block defined in the program:

1. Find the address of the QCB statement in the program compiler listing.

2. While the program is in the wait state, load $DEBUG in any available partition.

   Try to load $DEBUG from a terminal other than the terminal from which the waiting program was loaded. If you cannot use a different terminal, then load $DEBUG from the terminal used by the waiting program.

3. Enter the name of the waiting program when $DEBUG asks you for a program name and volume. Because the program is already loaded, you do *not* need to enter the volume name.

4. When $DEBUG asks for a partition, enter the number of the partition which contains the waiting program. If $DEBUG and the waiting program are in the same partition, press the enter key.

Analyzing the Instruction that Caused the Wait State *(continued)*

5. Reply **N** when asked if you want a new copy of the program loaded.

The following example shows what you would enter for the program WAITPGM located in partition 1. $DEBUG, in this example, is loaded in partition 2:

```
> $L $DEBUG
LOADING $DEBUG     31P,00:00:00, LP=B600, PART=2
PROGRAM (NAME,VOLUME): WAITPGM
PARTITION (DEFAULT IS CURRENT PARTITION): 1
ALREADY ACTIVE AT B400
DO YOU WANT A NEW COPY TO BE LOADED? N
```

6. Press the attention key and enter the **LIST** command.

7. Respond to the prompts to display the 5-word queue control block. For example, if the address of the QCB statement were at X'05E8', you would respond to the prompts as follows:

```
> LIST
OPTION (*/ADDR/R0...R7/#1/#2/IAR/TCODE/UNMAP): ADDR
ADDRESS: 5E8
LENGTH: 5
MODE(X/F/D/A/C): X
```

An example of the output follows:

```
05E8 X'0000 0000 0000 CD38 0001'
```

8. Look at word 3 of the queue control block. (The first word of the QCB is word 0.) Word 3 contains the task control block (TCB) address of the task that owns the QCB. In the sample output, the TCB address is X'CD38'. Word 4 contains the address space in which that task resides. Word 4 in the example shows address space 1 (partition 2).

9. Examine the task at the address (identified in step 8) and determine why that task did not issue a DEQ instruction.

The section "Common Causes of a Program Wait Using QCBs" on page PD-38 presents some hints as to what might be the cause of the problem.

10. Press the attention key and enter **END** to end $DEBUG.

Examining a Queue Control Block Defined in $SYSCOM

Do the following steps to examine the queue control block defined in $SYSCOM:

1. Using the link map listing of the current supervisor, find the address of the queue control block in $SYSCOM that you attempted to enqueue.

2. Press the attention key and enter **$CP 1**.

3. Press the attention key and enter **$D**.

4. Enter **0000** as the origin. Enter the queue control block address from step 1. Enter the number 5 for the count.

The following is an example of the output displayed for a queue control block at address X'19D0':

```
19D0: 0000 CD38 0000 1F00 0001
```

The first word of the QCB (word 0) indicates the status of the QCB. The value X'FFFF' means that the QCB is available. A value of X'0000' means that the QCB is enqueued upon.

5. Look at words 3 and 4 of the QCB. Word 3 is the task control block (TCB) address of the task that owns the QCB. In the sample output, this TCB address is X'1F00'. Word 4 contains the address space in which that task resides. In the sample output, the address space in which that task resides is address space 1 (partition 2).

   Word 1 contains the TCB address of the waiting task. Word 2 contains the address space in which that task resides. The waiting task is at address X'CD38" in address space 0 (partition 1).

6. Press the attention key and enter **$CP**, specifying the partition number you identified in step 5.

7. Press the attention key and enter **$A**.

Analyzing the Instruction that Caused the Wait State *(continued)*

8. Find the program whose load point is within the range of the TCB address you identified in step 5 on page PD-37.

Note: If the $A shows that no programs are active, the task whose TCB address you identified in step 5 on page PD-37 is no longer in storage and failed to issue a DEQ. When this is the case, you must IPL the system to clear the wait state and to release the enqueued QCB.

To prevent this condition in the future, determine what other programs use that QCB. If possible, also determine which of those programs was previously active. Examine those programs and determine which one failed to dequeue the QCB. The section "Common Causes of a Program Wait Using QCBs" presents some hints as to what might have caused the problem.

9. Subtract the program load point address from the TCB address of the task that owns the QCB. In this example, the TCB address is X'1F00'.

10. Using the resulting address from step 9, locate that address in the compiler listing for that program.

11. If that address points to an ENDPROG, ENDTASK, or DETACH statement, examine that program and determine why it did not issue a DEQ.

12. If that address does not point to an ENDPROG, ENDTASK, or DETACH statement, then the program in storage is not the program that enqueued the QCB. When this is the case, you must IPL the system to clear the wait state and to release the enqueued QCB.

To prevent this condition in the future, determine what other programs use that QCB. If possible, also determine which of those programs was previously active. Examine those programs and determine which one failed to dequeue the QCB. The section "Common Causes of a Program Wait Using QCBs" presents some hints as to what might have caused the problem.

Common Causes of a Program Wait Using QCBs

Wait states are often caused when:

- A program fails to issue a DEQ to an enqueued QCB.
- A program issues an ENQ to a queue control block defined in $SYSCOM when $SYSCOM is not mapped in that program's partition. You map $SYSCOM across partitions during system generation (COMMON= operand on the SYSTEM statement).

Analyzing the Instruction that Caused the Wait State *(continued)*

If $SYSCOM is not mapped in the partition in which you issued the ENQ or DEQ, ensure you use cross-partition services to enqueue or dequeue the QCB. Also check that the field $TCBADS of the program's TCB points to the address space in which the QCB resides. This consideration applies to any QCB not residing in a program's partition. See the *Language Reference* for examples of cross-partition operations.

- A program overlays the QCB area in storage (QCB destroyed).

Review the compiler listing of your program and ensure none of the previous conditions exist.

Analyzing an ENQT Instruction

When the program is pointing to an ENQT instruction, you must examine the terminal control block (CCB) of the device the program tried to enqueue. By examining the terminal control block, you can determine which task has control of that device.

Do the following steps to examine the terminal control block:

1. In the compiler listing, find the name of the terminal to which the program issued the ENQT.
2. Look in the link map listing of your current supervisor and locate the section labeled $EDXDEF. In that section, find the label that matches the name of the device the program tried to enqueue.
3. Add X'60' to the address of that device. The resulting address points to word 3 of the field $CCBQCB in the terminal control block.
4. At the terminal, press the attention key and enter **$CP 1**.
5. Press the attention key and enter **$D**.
6. Enter **0000** as the origin. Enter the address you calculated in step 3. Enter the number 2 for the count.
7. The first word displayed is the task control block (TCB) address of the program that has control of the device. The partition in which that program is running is the value of the second word plus 1.
8. Press the attention key and enter **$CP**, specifying the partition number from step 7.
9. Press the attention key and enter **$A**.

Analyzing the Instruction that Caused the Wait State *(continued)*

10. The TCB address from step 7 on page PD-39 will be within the range of the load point address for the program that has control of the device.

11. Examine the compiler listing of that program and determine why it has not issued a DEQT.

Analyzing a WAIT Instruction

If the event control block the program is waiting on is defined with an ECB statement, go to the section "Common Causes of a Program Wait Using ECBs" on page PD-41 for some hints as to what might be the problem.

If the event control block the program is waiting on is defined as a result of coding the EVENT= operand on a PROGRAM or TASK statement, do the following:

1. While the program is in the wait state, load $DEBUG in any partition.

Try to load $DEBUG from a terminal other than the terminal from which the waiting program was loaded. If you cannot use a different terminal, then load $DEBUG from the terminal used by the waiting program.

2. Enter the name of the program which contains the EVENT= operand when $DEBUG asks you for a program name and volume. Because the program is already loaded, you do *not* need to enter the volume name.

3. When $DEBUG asks for a partition, enter the number of the partition which contains the waiting program. If $DEBUG and the waiting program are in the same partition, press the enter key.

4. Reply **N** when asked if you want a new copy of the program loaded.

The following example shows what you would enter for the program WAITPGM located in partition 1. $DEBUG, in this example, is loaded in partition 2:

```
> $L $DEBUG
LOADING $DEBUG      31P,00:00:00, LP=B600, PART=2
PROGRAM (NAME,VOLUME): WAITPGM
PARTITION (DEFAULT IS CURRENT PARTITION): 1
ALREADY ACTIVE AT B400
DO YOU WANT A NEW COPY TO BE LOADED? N
```

5. Press the attention key and enter the **WHERE** command.

Analyzing the Instruction that Caused the Wait State *(continued)*

6. Using the compiler listing of that program, locate the instruction address displayed in step 5 on page PD-40 and determine why that program has not ended.

7. Press the attention key and enter **END** to end $DEBUG.

The next section, "Common Causes of a Program Wait Using ECBs," gives some hints as to what might be the problem.

Common Causes of a Program Wait Using ECBs

Wait states are often caused when a program:

- Fails to post an event control block (ECB) which another program is waiting on. Ensure that all attached tasks post the ECB before issuing a DETACH.
- Issues a WAIT with the RESET operand specified when the event has already been posted. Coding a WAIT followed by a RESET instruction may resolve the problem.
- Waits on an ECB defined in $SYSCOM when $SYSCOM is not mapped in the program's partition. You map $SYSCOM across partitions during system generation (COMMON= operand on the SYSTEM statement).

  If $SYSCOM is not mapped in the partition in which you issued the WAIT or POST, ensure you use cross-partition services to wait or post the ECB. Also check that the field $TCBADS of the program's TCB points to the address space the ECB resides. This consideration applies to any ECB not residing in a program's partition. See the *Language Reference* for examples of cross-partition operations.
- Has a logic error that unintentionally branches to a WAIT instruction.

Review the compiler listing of your program and ensure none of the previous conditions exist.

Analyzing the Instruction that Caused the Wait State *(continued)*

Other Possible Causes of a Wait State

When the program stops at an instruction other than ENQ, ENQT, or WAIT, consider the following:

- Is the program waiting for operator input to instructions such as READTEXT, GETVALUE, or QUESTION? The problem may be that the operator never responded to a prompt message or a prompt message requesting input was not coded.

- Is the instruction a READ or WRITE? It is possible that a hardware problem on disk prevented a device interrupt being sent to the supervisor. The system would wait until it received the device interrupt signaling completion of the I/O request.

  Any of the following may verify that a disk problem exists:

  - Verifying the disk using $INITDSK (VD command). If $INITDSK indicates errors, load $DASDI and try assigning alternate sectors on the device.
  - Allocating a data set using $DISKUT1.
  - Verifying the hardware configuration using $IOTEST (LS or LD command).
  - Sending messages to another terminal using $TERMUT3.

  If any or all of these attempts fail, the disk probably has a hardware problem. Contact your service representative for corrective action.

- Is a program, while using full screen support, enqueued to $SYSLOG? If the supervisor is unable to display a program check message to $SYSLOG, the system enters a wait state.

Chapter 6. Analyzing and Isolating a Program Check

The system issues a program check message to provide you with status information on an error that occurred during processing. This message is written to the terminal defined as $SYSLOG.

The system provides two types of program check messages: one for a system program check and one for an application program check. Application program checks are caused by errors within an application program. System program checks typically occur when the supervisor detects an error in its own code or when an application program somehow overlays part of the supervisor.

This chapter explains how to analyze the status information in a program check message so that you can determine the cause of a problem. A sample program that causes a program check when executed is included to show the steps required to isolate an error.

The first step in determining the cause of the problem is understanding the information displayed in the message. The following section explains the program check message.

Analyzing and Isolating a Program Check

How to Interpret the Program Check Message

The program check message can be in one of the following three formats:

- The standard format issued by the supervisor for application and all system program checks. The system issues the standard program check message for application programs when you do not code the ERRXIT= operand on the PROGRAM or TASK statement. Go to the section "Interpreting the Standard Program Check Message" when you receive the standard program check message.

- The format displayed when you code the ERRXIT= operand on the PROGRAM or TASK statement and specify the task error exit routine $$EDXIT. Refer to the *Event Driven Executive Language Programming Guide* for details on how to use $$EDXIT. Go to the section "Interpreting the Program Check Message from $$EDXIT" on page PD-50 when you receive this application program check message.

- Any format you create when you code the ERRXIT= operand on the PROGRAM or TASK statement and supply your own error exit routine. Refer to the *Customization Guide* for details on how to provide your own task error exit routine.

Interpreting the Standard Program Check Message

This section explains the information displayed in the standard program check messages. A description of the information follows the sample messages.

The following is an example of the standard application program check message:

```
PROGRAM CHECK:
 PLP  TCB  PSW  IAR  AKR  LSR  R0   R1   R2   R3   R4   R5   R6   R7
 3A00 0120 8002 2AD6 0110 80D0 0064 3B0A 3B20 3A37 3A34 015C 00B8 0000
```

The next example shows the system program check message:

```
SYSTEM PGM CHECK:
PSW  IAR  AKR  LSR  R0   R1   R2   R3   R4   R5   R6   R7
8002 2AD6 0110 80D0 0064 3B0A 3B20 3A37 3A34 015C 00B8 0000
```

The 11 words of information beginning with IAR and ending with R7 is called the level status block (LSB).

The headings displayed in the message and what the information means follows. (Normally when you analyze an EDL application program check, you need only be concerned with PLP, TCB, PSW, R1, R3, and R4.)

How to Interpret the Program Check Message *(continued)*

PLP — The address in storage of the program load point. This is the address at which the program was loaded for execution and represents the first word of your program listing.

For a system program check message, this field is omitted because the failing instruction is within the supervisor.

TCB — The address of the active task control block (TCB) as per the compiler listing (nonrelocated).

For a system program check message, this field is omitted because the failing instruction is within the supervisor.

PSW — The value of the processor status word (PSW) when the program check occurred. Refer to the section "How to Interpret the Processor Status Word" on page PD-47 to determine the meaning of this value.

IAR — The contents of the instruction address register (IAR) at the time of the error. The value shown is the address of the machine instruction currently executing.

AKR — The value of the address key register (AKR) at the time of the error. This last 3-hexadecimal digits indicate in which address space operand 1, operand 2, and the IAR reside. Bit 0 of the AKR is the equate operand spaces (EOS) bit. If bit 0 is set to 1, the address space key indicated for operand 2 is the address space key used for both operand 1 and operand 2.

LSR — The value of the level status register (LSR) when the error occurred. The bits, when set, indicate the following:

- Bits 0–4 — The status of arithmetic operations. Refer to the processor description manual for the meanings of these bits.
- Bit 8 — Program is in supervisor state.
- Bit 9 — Priority level is in process.
- Bit 10 — Class interrupt tracing is active.
- Bit 11 — Interrupt processing is allowed.

Bits 5–7 and bits 12–15 are not used and are always zero.

How to Interpret the Program Check Message *(continued)*

The next portion of the program check message displays the contents of the general purpose registers R0–R7. If the failing program were written in a language other than EDL, refer to the user's guide for that language to determine the register usage.

R0 Because the supervisor uses this register as a work register, the contents are usually not significant to the failing program.

R1 The address of the failing EDL instruction.

R2 The address in storage of the active task control block (TCB). The address in R2 is the sum of the TCB address and the load point address.

R3 The address in storage of EDL operand 1 of the failing instruction.

R4 The address in storage of EDL operand 2 (if applicable) of the failing instruction.

R5 The EDL operation code of the failing instruction. The first byte contains flag bits which indicate how operands are coded. For example, the flag bits indicate whether the operand is in #1, #2, or specified as a constant. The second byte is the operation code of the EDL instruction.

R6 Because the supervisor uses this register as a work register, the contents are usually not significant to the failing program. However, you can determine if the system was emulating EDL code when the failure occurred if R6 is twice the value shown in the second byte of R5. For example, if the second byte of R5 contained X'32' *and* the system was emulating EDL, R6 would contain X'0064'.

R7 Because the supervisor uses this register as a work register, the contents are usually not significant to the failing program. Sometimes the supervisor uses this register for a branch and link instruction. The address may give a clue as to which function passed control to the address in the IAR.

After reviewing the information shown in the program check message, you must analyze the contents displayed for the *processor status word* (PSW).

The processor status word is a 16-bit register the system uses to save error status. By looking at the processor status word, you can determine whether the error is hardware or software related. The next section explains how to interpret the processor status word.

How to Interpret the Processor Status Word

The value of the processor status word is shown as 4 hexadecimal digits. Each hexadecimal digit represents the sum of 4 binary bits. Starting from left to right, the value of each bit (when set) is 8, 4, 2, and 1. Thus to interpret what bits are on, you must convert each hexadecimal digit to binary. For example, if the PSW indicated the value X'8002', the binary representation and the bit positions would be as shown in Figure 6:

Hex value	Binary value	PSW bits
8	1000	0-3
0	0000	4-7
0	0000	8-11
2	0010	12-15

Figure 6. Sample Processor Status Word Bit Settings

In the previous example, note that bits 0 and 14 are set. These bit settings are the same as X'8002'.

After you convert the value to binary and identify which bit positions are set, refer to "Interpreting the Processor Status Word Bits" for an explanation of what each bit indicates. Remember that bit 0 is the leftmost bit in the 16-bit string.

Interpreting the Processor Status Word Bits

The information indicated by the processor status word bits can be categorized into three types:

- Software problems — bits 0–6
- Hardware problems — bits 8, 10, or 11
- Processor status — bit 7 and bits 12–15.

Figure 7 on page PD-48 shows the PSW bits and their general assignment for the different processors. An explanation of the bit settings follows Figure 7.

Refer to the specific processor description manual for details on class interrupts, I/O interrupts, and the basic instruction set (including indicator settings and possible exceptions conditions).

If the PSW indicates a hardware error (machine check), call your service representative for corrective action.

If the PSW indicates a software problem *and* the program check occurred in an application program, read the section "How to Analyze an Application Program Check" on page PD-54.

How to Interpret the Program Check Message *(continued)*

Review the section "How to Analyze a System Program Check" on page PD-67 if the error is a system program check.

Bit	Processor type 495x 2	3	4	5	6	Condition	Class interrupt
0	X	X	X	X	X	Specification check	Program check
1	X	X	X	X	X	Invalid storage address	Program check
2	X	X	X	X	X	Privilege violate	Program check
3	X		X	X	X	Protect check	Program check
4	X	X	X	X	X	Invalid function	Soft-exception
5			X	X	X	Floating-point exception	Soft-exception
6	X	X	X	X	X	Stack exception	Soft-exception
7						Extended Address Mode	None
8	X	X	X	X	X	Storage parity check	Machine check
9						Not used	
10	X	X	X	X	X	Processor control check	Machine check
11	X	X	X	X	X	I/O check	Machine check
12	X	X	X	X	X	Sequence indicator	None
13	X	X	X	X	X	Auto IPL	None
14	X		X	X	X	Translator enabled	None
15	X	X	X	X	X	Power/thermal warning	Power/thermal

Figure 7. Processor Status Word Bit Assignments

Processor Status Word Bit Descriptions

An explanation of the bit settings follows.

Bit 0 - Specification Check: Set to 1 if (1) the storage address violates the boundary requirements of the specified data type, or (2) the effective (computed) address is odd.

This error would occur, for example, if a program attempted to do a word move to an area on an odd-byte boundary. You can identify which operand (R3 or R4 addresses) violates the boundary if the last hex digit of the operand address is either 1, 3, 7, 9, B, D, or F.

This is a software error.

Bit 1 - Invalid Storage Address: Set to 1 when an attempt is made to access a storage address outside the storage size of the partition or when an attempt is made to refer to a storage address in a nonexistent partiton.

This error would occur, for example, if a program attempted to do a cross-partition move to a nonexistent partition.

This is a software error.

Bit 2 - Privilege Violate: Set to 1 if a program in problem state attempts to issue a privileged instruction. The processor can run in either supervisor or problem state. Some assembler instructions can be used only while in supervisor state. If an assembler program in problem state attempts to issue a privileged instruction, the privilege violate condition occurs.

Normally, this error would never occur in an EDL program.

This is a software error.

Bit 3 - Protect Check: Set to 1 if a program attempts to access protected storage. The processor can control access to areas in storage by using a storage protect feature. If a program attempts to address any part of the protected storage, the protect check indicator is set.

Normally, this error would never occur in an EDL program.

This is a software error.

Bit 4 - Invalid Function: Set to 1 by if any of the following conditions occur:

- Attempted execution of an illegal operation code or function combination.
- The processor attempts to execute an instruction associated with a feature that is not contained in the supervisor.

An EDL program can cause this error attempting to use floating-point instructions (FADD, FSUB, FMULT, or FDVID) when the floating-point support is not in the supervisor.

This is a software error.

Bit 5 - Floating-Point Exception: Set to 1 when an exception condition is detected by the optional floating-point processor. Floating-point hardware sets this bit to indicate underflow, overflow, and divide check exceptions. An EDL program can detect these exceptions by the return code from a floating-point instruction. No program check message is issued when this exception occurs.

This is a software error.

Bit 6 - Stack Exception: Set to 1 when an attempt has been made to pop an operand from an empty processor storage stack or push an operand into a full processor storage stack. A stack exception also occurs when the stack cannot contain the number of words to be stored by an assembler Store Multiple (STM) instruction.

Normally, this error would never occur in an EDL program.

This is a software error.

How to Interpret the Program Check Message *(continued)*

Bit 7 - Extended Address Mode: Set to 1 when the processor is operating in Extended Address Mode.

This is a status indicator.

Bit 8 - Storage Parity: Set to 1 when the hardware detects a parity error on data being read out of storage by the processor.

This is a hardware error.

Bit 10 - Processor Control Check: Set to 1 if no levels are active but execution continues.

This is a hardware error.

Bit 11 - I/O Check: Set to 1 when a hardware error has occurred on the I/O interface that may prevent further communication with any I/O device.

This is a hardware error.

Bit 12 - Sequence Indicator: Set to 1 to reflect the last I/O interface sequence to occur. This indicator is used in conjunction with I/O check (bit 11).

This is a status indicator.

Bit 13 - Auto IPL: Set to 1 by the hardware when an automatic IPL occurs.

This is a status indicator.

Bit 14 - Translator Enabled: Set to 1 when the Storage Address Relocation Translator Feature is installed and enabled.

This is a status indicator.

Bit 15 - Power Warning and Thermal Warning: Set to 1 when these conditions occur (refer to the appropriate processor manual for a description of a power/thermal warning class interrupt).

This is a status indicator.

Interpreting the Program Check Message from $$EDXIT

When you specify $$EDXIT as the task error exit for an EDL program, the output you receive is formatted with descriptive headings. In addition, $$EDXIT provides more information than the standard program check message. $$EDXIT also interprets the processor status word and tells you what it means.

How to Interpret the Program Check Message *(continued)*

When a program check occurs, the program check message is directed to $SYSLOG and $SYSPRTR.

The following is an example of a program check message issued by $$EDXIT. An explanation of each numbered item in the sample output follows the example.

```
        ******************************************
        * WARNING!!  AN EXCEPTION HAS OCCURRED!! *
        ******************************************

 1  PROGRAM NAME          = PCHECK   2  PSW = 8002
 3  PROGRAM VOLUME        = MYVOL    4  IAR = 2AD6
 5  PROGRAM LOAD POINT    =   0000   6  AKR = 0110
 7  ADDRESS OF ACTIVE TCB =   0120   8  LSR = 80D0
 9  ADDRESS OF CCB        =   0F5E  10  R0 (WORK REGISTER) =  0064
11  NUMBER OF DATA SETS   =      0  12  R1 (EDL INSTR ADDR) = 010A
13  NUMBER OF OVERLAYS    =      0  14  R2 (EDL TCB ADDR)   = 0120
15  $TCBADS               =   0001  16  R3 (EDL OP1 ADDR)   = 0037
17  ADDRESS OF FAILURE              18  R4 (EDL OP2 ADDR)   = 0034
     ((REL. TO PGM LOAD PT) = 010A  19  R5 (EDL COMMAND)    = 015C
20  DUMP OF FAIL ADDRESS            21  R6 (WORK REGISTER)  = 00B8
     010A: 015C 0000 0034 8332      22  R7 (WORK REGISTER)  = 0000
23  $TCBCO  =    -1 DEC;  FFFF HEX  24  #1 =  0037
25  $TCBCO2 =     0 DEC;  0000 HEX  26  #2 =  0000
27  PSW ANALYSIS:

       SPECIFICATION CHECK
       TRANSLATOR ENABLED
```

After this message is issued, $$EDXIT displays the following message on the loading terminal:

```
A MALFUNCTION HAS OCCURRED -- CALL SYSTEM PROGRAMMER
```

The previous message is not displayed if you code an extension error routine to $$EDXIT with the entry point name PCHKRTN. Refer to the *Customization Guide* for details on how to code an extension to $$EDXIT.

How to Interpret the Program Check Message *(continued)*

A description of the sample program check message follows.

1 The **PROGRAM NAME** field identifies the name of the failing application program. In this example, the program PCHECK failed.

2 The **PSW** field indicates the value of the *processor status word* when the error occurred. $$EDXIT interprets this value and displays its meaning as shown in field **27** of this sample message.

A detailed description of the processor status word and the associated bits are presented in the section "Interpreting the Processor Status Word Bits" on page PD-47.

3 The **VOLUME NAME** field identifies the name of the volume from which the failing application program was loaded. In this example, the name of the volume is MYVOL.

4 The **IAR** field (instruction address register) contains the address of the currently executing machine instruction.

5 The **PROGRAM LOAD POINT** field contains the address at which the program was loaded for execution. The address represents the first word of your program listing.

6 The **AKR** field contains the value of the address key register (AKR). The last 3-hexadecimal digits indicate in which address space operand 1, operand 2, and the IAR reside. Bit 0 of the AKR is the equate operand spaces (EOS) bit. If bit 0 is set to 1, the address space key indicated for operand 2 is the address space key used for both operand 1 and operand 2.

7 The **ADDRESS OF THE ACTIVE TCB** field contains the address (nonrelocated) of the active task control block (TCB) as per the compiler listing.

8 The **LSR** field level status register (LSR) information. The bits, when set, indicate the following:

- Bits 0–4 — The status of arithmetic operations. Refer to the processor description manual for the meanings of these bits.
- Bit 8 — Program is in supervisor state.
- Bit 9 — Priority level is in process.
- Bit 10 — Class interrupt tracing is active.
- Bit 11 — Interrupt processing is allowed.

Bits 5–7 and bits 12–15 are not used and are always zero.

How to Interpret the Program Check Message *(continued)*

9 The **ADDRESS OF CCB** field contains the address of the terminal control block (CCB) assigned to the failing program.

10 The **R0** field contains the contents of hardware register 0 when the error occurred. Because the supervisor uses this register as a work register, the contents are usually not significant when you analyze the failing program.

11 The **NUMBER OF DATA SETS** field shows the number of data sets specified on the DS= operand of the PROGRAM statement.

12 The **R1** field contains the address of the failing EDL instruction.

13 The **NUMBER OF OVERLAYS** field indicates the number of overlay programs specified on the PGMS= operand of the PROGRAM statement.

14 The **R2** field contains the address in storage of the active task control block. This address is the sum of the TCB address and the program load point.

15 The **$TCBADS** field contains the target task address space. The value of this field plus 1 indicates the partition number in which the program was running.

16 The **R3** field contains the address of EDL operand 1 for the failing EDL instruction.

17 The **ADDRESS OF FAILURE** field contains the address of the failing EDL instruction. This is the address shown in the compiler listing. This is also the address shown in field **12** in this sample output. In this example, the failing EDL instruction is at address X'010A'.

18 The **R4** field contains the address of EDL operand 2 (if applicable) for the failing EDL instruction.

19 The **R5** field contains the EDL operation code of the instruction that was executing when the failure occurred. The first byte contains flag bits which indicate how operands are coded. For example, the flag bits indicate whether the operand is in #1, #2, or specified as a constant. The second byte is the operation code of the EDL instruction.

20 The **DUMP OF FAIL ADDRESS** field shows the location and content of the instruction that was executing when the failure occurred. The information at this address also appears in the compiler listing.

21 The **R6** field contains the contents of hardware register 6 when the error occurred. Because the supervisor uses this register as a work register, the contents are usually not significant when you analyze the failing program. However, you can determine if the system was emulating EDL code when the failure occurred if R6 is twice the value shown in the second byte of R5. For example, if the second byte of R5 contained X'32' *and* the system was emulating EDL, R6 would contain X'0064'.

How to Interpret the Program Check Message *(continued)*

22 The **R7** field contains the contents of hardware register 7 when the error occurred. Because the supervisor uses this register as a work register, the contents are usually not significant when you analyze the failing program.

Sometimes the supervisor uses this register for a branch and link instruction. The address may give you a clue as to which function passed control to the address in the IAR.

23 The **$TCBCO** field shows the value in the first word of the failing program's task control block (TCB). The value is displayed in decimal and followed by the hexadecimal equivalent.

24 The **#1** field shows the contents of index register 1 when the failure occurred. In this example, #1 contains the value X'0037'.

25 The **$TCBCO2** field shows the value in the second word of the failing program's task control block (TCB). The value is displayed in decimal and followed by the hexadecimal equivalent.

26 The **#2** field shows the contents of index register 2 when the failure occurred.

27 The **PSW ANALYSIS** field explains the meanings of the bit settings in the processor status word (PSW). The hexadecimal format of the processor status word is shown in field **2**. This information indicates the type of error that occurred.

Refer to the section "Processor Status Word Bit Descriptions" on page PD-48 to determine the type of error the "PSW ANALYSIS" field indicates.

If the error points to hardware, call your service representative for corrective action.

If the error points to software, read the following section.

How to Analyze an Application Program Check

When the processor status word (PSW) indicates a software error, you need to find out where in the program the error occurred. The information in the program check message can help you find the error.

Presented in this section is a sample program check message and the program that caused the program check. Using both the program check message and the compiler listing of the sample program, this section will explain the steps required to find the problem. The techniques used can help you to isolate program checks in your application programs. The section "Examining an Unmapped Storage Area for the Cause of a Program Check" on page PD-60 presents techniques that may be helpful if your program uses unmapped storage.

The section "Some Common Causes of Application Program Checks" on page PD-66 provides some additional hints about what may cause this type of error.

How to Analyze an Application Program Check *(continued)*

To find the cause of the program check, do the following:

1. Look at the program check message and determine what type of software error the processor status word indicates.

The program check message from the sample program follows:

```
PROGRAM CHECK:
 PLP  TCB  PSW  IAR  AKR  LSR  R0   R1   R2   R3   R4   R5   R6   R7
 3A00 0120 8002 2AD6 0110 80D0 0064 3B0A 3B20 3A37 3A34 015C 00B8 0000
```

The PSW indicates that a specification check occurred and that the translator was enabled. A specification check indicates a boundary violation. Thus, the specification check is the cause of the error.

2. Look at the addresses for operands 1 and 2 and determine which operand is on an odd-byte boundary. R3 contains the address of operand 1. R4 contains the address of operand 2.

Determining which operand is on an odd-byte boundary can help you analyze the failing instruction.

In the sample program check message, notice that the address of operand 1 (X'3A37') is on an odd-byte boundary.

3. Find the address of the failing instruction. Subtract the program load point (PLP) from the address of R1. The result is the address of failing instruction.

The program load point of the sample program is X'3A00'. The value of R1 is X'3B0A'. The result of subtracting these addresses is X'010A'.

At this point you know the address of the failing instruction and which operand is on an odd-byte boundary.

4. Look in the compiler listing and determine if the instruction at the address you calculated in step 3 is coded correctly.

How to Analyze an Application Program Check *(continued)*

In the compiler listing of the sample program, a MOVE instruction is at address X'010A':

```
LOC   +0   +2   +4   +6   +8
0000  0008 D7D9 D6C7 D9C1 D440  PCHK   PROGRAM   START
000A  0000 0120 01A0 0000 0000
0014  01A4 0000 0000 0000 0100
001E  01A2 0000 0000 0000 0000
0028  0000 0000 0000 0000 0000
0032  0000
0034  4040                      A      DATA      X'4040'
0036  0000 0000 0000 0000 0000  B      DATA      100F'0'
00FE                            START  EQU       *
00FE  835C 0000 0036                   MOVEA     #1,B
0104  809C 0116 0064                   DO        100
010A  015C 0000 0034                     MOVE    (0,#1),A
0110  8332 0000 0001                     ADD     #1,1
0116  009D 0000 0001                   ENDDO
011C  0022 FFFF                        PROGSTOP
0120  0000 0000 0000 0234 0000         ENDPROG
012A  00D0 0000 00FE 0120 0000
0134  0000 0000 0000 0000 0000
013E  0002 0096 0000 0000 FFFF
0148  0000 0000 014C 0000 0000
0152  014E D7C3 C8D2 4040 4040
015C  0000 0000 0000 0000 0000
0166  0000 0000 FFFF 0000 0000
0170  0000 0000 0000 0120 0000
017A  0000 0000 0000 0000 0000
0198  0000 0000 0120 0080 0000
01A2  0000 0000 0000 0000 0000
01B6  0000
01B8                                   END
```

In this example, the MOVE instruction and its operands are coded correctly. Because the cause of the error is not apparent by looking the the failing instruction, you can use $DEBUG to trace the program's execution.

5. At the terminal, press the attention key and load $DEBUG. Enter the name of the program (and volume if not on EDX002) when $DEBUG asks you for the program name and volume.

When $DEBUG asks you for a partition, enter the number of the partition where you want the failing program to be loaded. If you want the program loaded in the same partition as $DEBUG, press the enter key. For the "TERMINAL" prompt, enter the terminal on which you want $DEBUG to load the program. If you press the enter key, $DEBUG loads the program on the terminal it is currently using.

How to Analyze an Application Program Check *(continued)*

In this example, $DEBUG is loaded in partition 2. The utility loads the failing program, PCHK, in the same partition and the program and the utility share the same terminal.

```
> $L $DEBUG
LOADING $DEBUG     31P,00:00:00, LP=0000, PART=2
PROGRAM (NAME,VOLUME): PCHK
PARTITION (DEFAULT IS CURRENT PARTITION):
TERMINAL (DEFAULT IS CURRENT TERMINAL):
LOADING PCHK        2P,00:00:00, LP=1F00, PART=2

REQUEST "HELP" TO GET LIST OF DEBUG COMMANDS
PCHK     STOPPED AT  00FE
```

6. Press the attention key and enter **AT** to set the first breakpoint at the address of the program's entry point (low address). Enter **TASK** when you are prompted for an option. The entry point in the sample program is at address X'00FE'. This sequence follows:

```
> AT
OPTION(*/ADDR/TASK/ALL): TASK
LOW ADDRESS: FE
```

7. Set the next breakpoint at the address of the last executable instruction (high address). The last executable instruction of the sample program is the PROGSTOP at address X'011C'.

Because you only need the trace addresses at this point, select the NOLIST and NOSTOP options:

```
HIGH ADDRESS: 11C
LIST/NOLIST: NOLIST
STOP/NOSTOP: NOSTOP
     1 BREAKPOINT(S) SET
```

8. Press the attention key and enter **GO**.

The program will run until it program checks again. During its execution, however, $DEBUG will display all the instruction addresses up to the point of the program check.

The following is an example of the trace addresses from the sample program:

```
PCHK     CHECKED AT  0104
PCHK     CHECKED AT  010A
PCHK     CHECKED AT  0110
PCHK     CHECKED AT  0116
PCHK     CHECKED AT  010A
```

9. Look at the trace addresses. Notice that in the sample trace output, the instruction at address X'010A' (MOVE) executed successfully the first time. However, the second time the program executed the instruction at X'010A', the program failed with a program check. The supervisor cancels the program.

Because the last instruction the program executed was at address X'010A', you need to reload the program under $DEBUG, set a breakpoint at address X'010A', and examine index register 1 (#1). The sample program uses the index of #1 to point to the target address of the MOVE instruction.

By examining #1 before the program executes the instruction at X'010A', you can determine if #1 points to an odd-byte boundary.

10. Press the attention key and enter **END** to end the current $DEBUG.

11. Reload $DEBUG and specify the name of the program.

12. Press the attention key and enter **AT**.

13. For the sample program, reply to the prompts as follows to set a breakpoint at address X'010A' and to examine #1:

```
OPTION(*/ADDR/TASK/ALL): ADDR
BREAKPOINT ADDR: 10A
LIST/NOLIST: LIST
OPTION( */ADDR/R0...R7/#1/#2/IAR/TCODE/UNMAP): #1
LENGTH: 1
MODE(X/F/D/A/C): X
STOP/NOSTOP: STOP
      1 BREAKPOINT(S) SET
```

14. Press the attention key and enter **GO**.

$DEBUG stops the program's execution at address X'010A' and displays the contents of #1. The following is an example of the output:

```
PCHK     STOPPED AT  010A
  #1 PCHK     X' 1F36'
```

The value X'1F36' in #1 is the address *in storage* of the variable labeled "B". This address gets stored in #1 on the previous MOVEA instruction. Notice that at this point, the address for operand 1 (#1) points to an even address (word aligned).

How to Analyze an Application Program Check *(continued)*

The trace output showed that no problem occurred the first time through the DO loop. Thus, you can assume that some instruction after that point caused the address in #1 to point to an odd-byte boundary.

The next sequence shows how you can identify the cause of the problem.

15. Press the attention key and enter **GO**.

Again $DEBUG stops the program's execution at address X'010A' and displays the contents of #1. The following sample output shows what #1 points to now:

```
PCHK      STOPPED AT  010A
  #1 PCHK      X' 1F37'
```

Notice that the address #1 points to is on an odd-byte boundary (X'1F37'). Further examination of the compiler listing shows that immediately after the MOVE instruction, the program incremented the value in #1 by 1:

```
                                              •
                                              •
                                              •
OOFE    835C 0000 0036                        MOVEA     #1,B
0104    809C 0116 0064                        DO        100
010A    015C 0000 0034                          MOVE    (0,#1),A
0110    8332 0000 0001                          ADD     #1,1
0116    009D 0000 0001                        ENDDO
```

Because the program attempts to move a word of data and #1 points to an odd-byte boundary (X'1F37'), the program fails with a specification check.

Although the program check message indicates that the MOVE instruction failed, the cause of the problem is the ADD instruction at address X'0110'.

Because the MOVE instruction attempts to move a word of data, the program should have incremented #1 by 2. Adding 2 to #1 enables the program to receive the next word of data on a word boundary.

How to Analyze an Application Program Check *(continued)*

Examining an Unmapped Storage Area for the Cause of a Program Check

An application program check can occur if a program receives invalid data. By using the LIST command of $DEBUG, you can examine the data areas in your program to see if any of the data in these areas is invalid. (For more information on using the LIST command of $DEBUG, refer to the *Operator Commands and Utilities Reference*.) If the failing program uses unmapped storage, you may also want to look at the data in the unmapped storage areas. This section explains how to examine an unmapped storage area to determine the cause of an application program check.

The sample program used in this section is named CODE. The CODE program reads a set of addresses into unmapped storage, acquires the data at those addresses, and processes the data. The last time CODE was loaded, however, the operator received a program check message. The program check message from the sample program follows:

```
PROGRAM CHECK:
 PLP  TCB  PSW  IAR  AKR  LSR  R0   R1   R2   R3   R4   R5   R6   R7
 0000 095A 4002 3CBA 0330 88D0 0080 08DE 0904 00A0 8210 025C 00B8 0000
```

The PSW in the message indicates that the sample program attempted to use an invalid storage address. This error can occur if a program attempts to use an address that is outside of the partition in which the program was loaded. It also can occur if a program refers to a storage address in a nonexistent partition. In addition to the software error, the PSW also shows that the translator was enabled. (See "Interpreting the Processor Status Word Bits" on page PD-47 for an explanation of the bit settings.)

To find the address of the failing EDL instruction, subtract the program load point (PLP) from the contents of R1 in the program check message. The value of R1 in the program check message is X'08DE'. Since the program load point for the sample program is X'0000', the address of the failing EDL instruction is X'08DE'.

How to Analyze an Application Program Check *(continued)*

In the compiler listing for the sample program, a MOVE instruction is at address X'08DE':

```
LOC    +0   +2   +4   +6   +8
0000   0008 D7D9 D6C7 D9C1 D440  CODE   PROGRAM   START,DS=((DATA,VOL))
000A   0000 0104 0184 0000 0000
0014   0188 0000 0001 0000 0100
001E   0186 0000 0000 0000 0000
0028   0000 0000 0000 0000 0000
0032   FFFF 0000 0000 0808 C4C1
003C   E3C1 4040 4040 0606 E5D6
0046   D340 4040 0000 0000 0000
0050   0000 0001 0000 0001 0000
005A   0000 0000 0000 0000 0000
006E   0000 0000 0000 0000
0076   0000 C1C1 0000 0000 0008  BLOCK  STORBLK   TWOKBLK=1,MAX=2
0080   0002 FFFF 0000 0000 0090
008A   0000 0000 0000 FFFF FFFF
0094   FFFF FFFF
0098   0000                      INDEX  DC        F'0'
009A   0000 0000 0000 0000 0000  ENTRY  DC        1024F'0'
0892   0000 0000 0000 0000
089A                             START  EQU       *
089A   00B9 0076 0000 0000 0101         GETSTG    BLOCK,TYPE=ALL
08A4   80B9 0076 0001 0000 0300         SWAP      BLOCK,1
08AE   035C 0000 0082                   MOVE      #1,BLOCK+$STORMAP
08B4   8120 0000 0008 0000 020C         READ      DS1,(0,#1),8
08BE   0032
08C0   835C 0002 009A                   MOVEA     #2,ENTRY
08C6   805C 0098 0000 809C 08F0         DO        128,TIMES,INDEX=INDEX
08D0   0080 8032 0098 0001
08D8   045C 08E2 0000                     MOVE      ADDRESS,(0,#1)
08DE   025C 0000 08E2                     MOVE      (0,#2),*,P2=ADDRESS
08E4   8332 0000 0002                     ADD       #1,2
08EA   8332 0002 0002                     ADD       #2,2
08F0   009D 0000 0001                   ENDDO
                                          •
                                          •
                                          •
0946   00B9 0076 0000 0000 0201         FREESTG   BLOCK,TYPE=ALL
0A00   0022 FFFF                        PROGSTOP
                                        COPY      STOREQU
                                          •
                                          •
                                          •
```

The MOVE instruction at X'08DE' should take a word of data from an address in storage and place it in the data area labeled ENTRY at X'009A'. The address of ENTRY is contained in #2. The MOVE instruction moves data from addresses supplied by the previous MOVE instruction at X'08D8'. The addresses reside in the unmapped storage area obtained by the program.

From the program check message, it appears that the MOVE instruction at X'08DE' received a storage address that was not in the partition in which the program was loaded. To determine if this was the case, you first need to know the partition CODE was loaded in and the largest storage address in that partition.

How to Analyze an Application Program Check *(continued)*

In this example, the operator loaded CODE in partition 2. You can find the largest storage address in a partition by looking at the storage map for your system. The storage map appears on the last page of the listing created when you generated your system. It also is displayed when you IPL your system.

Look under the heading "TOTAL SIZE (HEX)" in the storage map and find the value listed for the partition. Subtract 1 from this value to get the largest usable storage address in the partition. For the sample system on which CODE is running, the storage map shows a total size for partition 2 of X'8000'. Therefore, the largest usable address in partition 2 is X'7FFF'.

To see if the sample program attempted to gain access to a storage address greater than X'7FFF', you need to look at the data in the unmapped storage area used by the program. To examine the contents of an unmapped storage area, do the following:

1. Load $DEBUG in any available partition.

2. Enter the name of the failing program (and volume if not on EDX002).

3. When $DEBUG asks for a partition, enter the number of the partition where the utility should load the failing program. If you want the program loaded in the same partition as $DEBUG, press the enter key.

4. For the "TERMINAL" prompt, enter the terminal on which you want $DEBUG to load the program. If you press the enter key, $DEBUG loads the program on the terminal it is currently using.

In the following example, $DEBUG is loaded in partition 1. The utility loads the sample program in partition 2, but $DEBUG and the program share the same terminal.

```
> $L $DEBUG
LOADING $DEBUG      45P,00:01:30, LP=B600, PART=1
PROGRAM (NAME,VOLUME): CODE
PARTITION (DEFAULT IS CURRENT PARTITION): 2
TERMINAL (DEFAULT IS CURRENT TERMINAL):
LOADING CODE         3P,00:01:32, LP=0000, PART=2

REQUEST "HELP" TO GET LIST OF DEBUG COMMANDS
CODE      STOPPED AT  089A
```

5. Press the attention key and enter **AT** to set a breakpoint at the address following the instruction that reads the data into unmapped storage.

   Note: If your program obtains several unmapped storage areas, you may need to trace the execution of the program to determine what area was in use when the program check occurred. Review the trace procedure beginning with step 6 on page PD-57.

In the sample program, the address following the READ instruction is X'08BE':

```
> AT
OPTION (*/ADDR/TASK/ALL): ADDR
BREAKPOINT ADDR: 8BE
LIST/NOLIST: NOLIST
STOP/NOSTOP: STOP
      1 BREAKPOINT(S) SET
```

6. Press the attention key and enter **GO**.

$DEBUG displays a message when it suspends the program's execution at the breakpoint:

```
CODE      STOPPED AT  08BE
```

7. Press the attention key and enter the **LIST** command. After you enter this command, do the following:

   a. For "OPTION", enter **UNMAP**.

   b. For "STORBLK ADDRESS", enter the address of the STORBLK statement that defines the unmapped storage area you want to see.

   c. For "SWAP#", enter the number of the unmapped storage area you want to see.

   d. For "DISPLACEMENT", indicate how far from the beginning of the unmapped storage area the utility should go before listing the contents of the area. Enter a number of bytes (in hexadecimal). For example, if you enter **1A**, $DEBUG will begin the listing after the 26th byte in the unmapped storage area.

   e. For "LENGTH", enter the number of words, doublewords, or characters you want to list, depending on the MODE you select. Enter a decimal number.

   f. For "MODE", enter the format you want the data to appear in.

How to Analyze an Application Program Check *(continued)*

The following example shows how you would list the first 256-byte record the sample program read into unmapped storage.

```
> LIST
OPTION (*/ADDR/RO...R7/#1/#2/IAR/TCODE/UNMAP): UNMAP
STORBLK ADDRESS (0 TO CANCEL LIST): 76
SWAP#: 1
DISPLACEMENT: 0
LENGTH: 128
MODE(X/F/D/A/C): X
```

Figure 8 shows how $DEBUG displays the first record of the unmapped storage area for the CODE program.

```
0000 X' 0B36 0B38 0B3A 0B3C 0B3E 0B40 0B42 0B44'
0010 X' 0B46 0B48 0B4A 0B4C 0B4E 0B50 0B52 0B54'
0020 X' 0B56 0B58 0B5A 0B5C 0B5E 0B60 0B62 0B64'
0030 X' 0B76 0B78 0B7A 0B7C 0B7E 0C00 0C02 0C04'
0040 X' 0C06 0C08 0C0A 0C0C 0C0E 0C10 0C12 0C14'
0050 X' 0C16 0C18 0C1A 0C1C 0C1E 0C20 0C22 0C24'
0060 X' 0C26 0C28 0C2A 0C2C 0C2E 0C30 0C32 0C34'
0070 X' 0D00 0D02 0D04 0D06 0D08 0D0A 0D0C 0D0E'
0080 X' 0D10 0D12 0D14 0D16 0D18 0D1A 0D1C 0D1E'
0080 X' 7200 7202 7204 7206 7208 720A 720C 720E'
0090 X' 8210 7212 7214 7216 7218 721A 721C 721E'
00A0 X' 7220 7222 7224 7226 7228 722A 722C 722E'
00B0 X' 2100 2102 2104 2106 2108 210A 210C 210E'
00C0 X' 2110 2112 2114 2116 2118 211A 211C 211E'
00D0 X' 2120 2122 2124 2126 2128 212A 212C 212E'
00E0 X' 2130 2132 2134 2136 2138 213A 213C 213E'
00F0 X' 2140 2142 2144 2146 2148 214A 214C 214E'
```

Figure 8. Sample Listing from $DEBUG

Notice the word of data at address X'0090' in Figure 8. The word contains the value X'8210'. When the MOVE instruction in the sample program attempted to use this value as an address, it went beyond the bounds of the partition and caused the program check.

How to Analyze an Application Program Check *(continued)*

To verify that the address caused the program check, you could replace it with a valid address (one smaller than X'7FFF') and see if the program runs successfully. You can replace data in an unmapped storage area with the PATCH command of $DEBUG.

To use the PATCH command:

1. Press the attention key and enter **PATCH**.

2. After you enter the command, do the following:

   a. For "OPTION", enter **UNMAP**.

   b. For "STORBLK ADDRESS", enter the address of the STORBLK statement that defines the unmapped storage area you want to modify.

   c. For "SWAP#", enter the number of the unmapped storage area you want to modify.

   d. For "DISPLACEMENT", indicate how far from the beginning of the unmapped storage area the utility should go before listing the contents of the area. Enter a number of bytes (in hexadecimal). For example, if you enter **1A**, $DEBUG will begin the listing after the 26th byte in the unmapped storage area.

   e. For "LENGTH", enter the number of bytes, up to 16, that you want to modify. You cannot modify more than 16 bytes of data at a time. Enter a decimal number.

   f. For "MODE", enter the format you want the data to appear in.

3. The PATCH command displays the data to be modified. Enter your new data following the "DATA:" prompt message. Separate each word of data with a space.

   If you enter less data than the amount displayed, the command pads the remaining area with blanks (for character data) or zeros (for all other types of data).

How to Analyze an Application Program Check *(continued)*

4. The command displays the data you entered and issues the prompt message "YES/NO/CONTINUE". Respond **Y** to confirm the change, **N** to cancel the change, or **CONTINUE** to confirm the change and to continue modifying data.

The following example uses the PATCH command to replace the invalid address in the unmapped storage area with the address X'7210'. After the change is made, program execution is resumed by pressing the attention key and entering **GO**.

```
> PATCH
OPTION (*/ADDR/RO...R7/#1/#2/IAR/TCODE/UNMAP): UNMAP
STORBLK ADDRESS (0 TO CANCEL PATCH): 76
SWAP#: 1
DISPLACEMENT: 90
LENGTH: 1
MODE(X/F/D/A/C): X
NOW IS
 0090 X' 8210'
DATA: 7210
NEW DATA
 0090 X' 7210'
YES/NO/CONTINUE: Y

> GO
```

Some Common Causes of Application Program Checks

Program checks in an application program are commonly caused by the following:

- PROGSTOP statement omitted in the program
- Failure to link-edit programs with external references (EXTRNs)
- Nonexecutable statements coded within inline executable code
- Attempting to move a word of data to an odd-byte boundary
- Reading or moving data into a storage area too small to contain the data.

How to Analyze a System Program Check

Generally a system program check is caused by either of the following:

- An error in the assembly or link-edit of the current supervisor during system generation.
- An application program that somehow overlays a part of the supervisor in storage.

This section describes some methods you may be able to use to isolate the cause of a system program check.

To begin analyzing the system program check, do the following:

1. Review the compiler and link-edit listings of the current supervisor for -1 completion codes. If either of the listings do not indicate successful completion, correct the errors and perform another system generation.

2. Try to reproduce the failure by rerunning all the programs that were active. Ensure those programs run in the same partition they were running in when the failure occurred. While you rerun the programs, identify which program caused the failure.

   A program that was running in a partition containing supervisor code or a program doing a cross-partition move is most likely the cause of the problem.

   After determining which program caused the failure, go to the section "Analyzing the Program Causing the System Program Check."

3. If you determine that the cause of the failure was not due to an application program, submit an authorized program analysis report (APAR) along with a stand-alone dump the next time the failure occurs.

Analyzing the Program Causing the System Program Check

The program you identified as the cause of the system program check probably overlaid an area of the supervisor. To correct the problem, you need to find the instruction in the program that overlays the supervisor area.

This section explains two techniques you can use to isolate the cause of the failure. The technique you use depends on the contents of the instruction address register (IAR) shown in the system program message.

If the address shown in the IAR does not contain all zeros, review the following section. Go to the section "Technique 2 — IAR is All Zeros" on page PD-69 when the IAR address is all zeros.

How to Analyze a System Program Check *(continued)*

Technique 1 — IAR is Non-Zero

To isolate the problem, do the following:

1. Record the address shown for the instruction address register (IAR) in the system program check message.

2. Press the Load button to re-IPL the system.

3. Press the attention key and enter **$CP 1**.

4. Press the attention key and enter **$D**.

5. Enter **0000** as the origin. Enter the IAR address from step 1. Enter the number 1 for the count.

6. Record the value displayed for that address.

7. Press the attention key and load $DEBUG.

8. Enter the name of the program you identified as the cause of the problem.

The next sequence of steps enable you to determine if the contents displayed in step 6 change during the program's execution. By setting breakpoints at various addresses in the program and determining when the value from step 6 changes, you can locate the portion of the program that causes the error.

9. Using the compiler listing of the program, select several addresses throughout the program at which you want $DEBUG to stop the program's execution.

10. Press the attention key and enter **AT**.

11. At the prompts, enter **ADDR**, a breakpoint address, and the **NOLIST** and **STOP** options.

12. Repeat steps 10 and 11 for each breakpoint address you selected.

13. Press the attention key and enter **GO**.

14. When $DEBUG stops the program's execution at the breakpoint, press the attention key and enter **$D** in partition 1.

15. Enter **0000** as the origin. Enter the IAR address from step 1. Enter the number 1 for the count.

16. Determine whether the value now displayed is the same value you recorded in step 6 on page PD-68.

17. Repeat steps 13 through 16 until you notice a value other than the value shown in step 6 on page PD-68. When you notice a different value, go to step 18.

18. In the compiler listing, look at the instructions between the last two breakpoint addresses. One or more of the instructions within those breakpoint addresses are the instructions that overlaid a supervisor area and caused a system program check.

19. Determine what instructions caused the failure and correct the error.

Technique 2 — IAR is All Zeros

This technique uses $DEBUG to trace the program's execution. To isolate the problem, do the following:

1. Press the attention key and enter **$CP 1**.

2. Press the attention key and load $DEBUG.

3. Enter the name of the program you identified as the cause of the problem.

4. Press the attention key and enter **AT** to set the first breakpoint at the address of the program's entry point. Enter **TASK** when $DEBUG prompts for an option. For the low address, enter the address of the program's entry point.

5. Enter the address of the program's last executable instruction as the high address.

6. Press the attention key and enter **GO**.

7. When the system program check occurs, the instruction that caused the failure is most likely at one of the last few addresses shown in the trace output.

8. Examine the compiler listing and determine which instruction caused the failure.

9. Correct the error and recompile the program.

Notes

Chapter 7. Analyzing a Failure Using a Storage Dump

This chapter explains how you can use a storage dump created by either $TRAP or the stand-alone dump method to analyze a failure. The discussions include how to analyze a wait state, run loop, and a program check.

Very often when you use a dump to analyze a failure, you may have to look at control blocks to find information about the failure. You can obtain a control block equate listing (copy code) by including a COPY statement in your program and specifying the name of the control block you need. The *Language Reference* contains a list of commonly used control block equate names. The control block equates reside on volume ASMLIB and end with the characters "EQU". The *Internal Design* shows the control blocks in detail.

Before you begin to analyze a failure using a dump, you need to know how to interpret the various fields shown in a dump and what they mean. The following section explains the various fields of a dump.

Analyzing a Failure Using a Storage Dump

Interpreting the Dump

This section explains the various fields of a sample dump. $TRAP was used to produce the sample dump presented in this section.

Some of the fields shown in a dump differ depending on whether you created the dump using $TRAP or the stand-alone dump method. These differences are noted in the explanation of the sample dump where appropriate. In addition, some of the fields that can appear in a dump depend on the devices and features installed on your system.

The examples presented show how $DUMP prints the information when you select the "format control block" option. The order in which the examples are presented is the same order the information would appear in a dump.

Note: If you are using a processor in Extended Address Mode, see the *Extended Address Mode and Performance Analyzer User Guide* for additional information on the format of the dump.

The various pieces of the dump shown in this section have numbered items. An explanation of the numbered items follows each example.

Hardware Level and Register Contents

Figure 9 shows the first part of the dump.

```
1 EVENT DRIVEN EXECUTIVE $TRAP FORMAT STORAGE DUMP

2 AT TIME OF TRAP PSW WAS 8006 ON HARDWARE LEVEL 1

3          LEVEL 0    LEVEL 1    LEVEL 2    LEVEL 3       SVC-LSB   SVCI-LSB
4 IAR         1FFA       2AD6       1F32       1F32          1F32       1F0A
5 AKR         0100       0110       0000       0000          0000       0000
6 LSR         8090       00D0       0090       0090          00C0       00C0
7 R0          0000       0001       0000       0000          0000       0000
  R1          0000       0044       0000       0000          0000       0000
  R2          02C2       02C2       0000       0000          0000       0000
  R3          02B6       004D       0000       0000          0000       0000
  R4          0000       0048       0000       0000          0000       0000
  R5          0001       805C       0002       0003          0001       0000
  R6          0000       00B8       8000       8000          8000       0000
  R7          0000       0000       0000       0000          0000       0000
```

Figure 9. Hardware Level and Register Contents

Item **1** as shown in Figure 9 on page PD-72 indicates what type of dump was taken. This example indicates a $TRAP dump. If a stand-alone dump were taken, the text **STAND ALONE STORAGE DUMP** would appear.

Item **2** indicates the value of the processor status word (PSW) and the active hardware interrupt level. In the sample dump, the PSW value indicates X'8006' on hardware level 1. A $TRAP dump always shows the value of the PSW and the active level; a stand-alone dump never contains this line of information.

Refer to the section "How to Interpret the Processor Status Word" on page PD-47 for the meaning of the processor status word.

The column headings at item **3** identify six level status blocks (LSB). There is an 11-word level status block shown for each of the system's hardware interrupt levels (0–3). In addition, the contents of the SVC (supervisor call) LSB and the SVCI (supervisor call immediate action) LSB are shown.

The contents of a level status block for a particular hardware interrupt level is shown vertically beginning with IAR and ending with R7. The fields shown for a level status block in the dump are also displayed in a program check message.

Level 0 is inaccurate in the stand-alone dump. This is the level on which the dump program runs; therefore, none of the information for level 0 in a stand-alone dump is relevant to the problem being analyzed. However, the information shown for level 0 in a $TRAP dump *is reliable*; $TRAP saves the information for level 0 as well as levels 1, 2, and 3.

EDX uses the four hardware levels as follows. Level 0 is the highest priority level:

Level 0 — Timer interrupts and task dispatcher

Level 1 — Attention list tasks, supervisor tasks, and I/O interrupts

Level 2 — EDL tasks with a priority of 1–255

Level 3 — EDL tasks with a priority of 256–510.

Interpreting the Dump *(continued)*

Item **4** shows the contents of the instruction address register (IAR). The value shown is the address of the machine instruction currently executing.

Item **5** shows the value of the address key register (AKR). The last 3-hexadecimal digits indicate in which address space operand 1, operand 2, and the IAR reside. Bit 0 of the AKR is the equate operand spaces (EOS) bit. If bit 0 is set to 1, the address space key indicated for operand 2 is the address space key used for both operand 1 and operand 2.

The value of the AKR for level 1 in the sample dump (X'0110') indicates operands 1 and 2 reside in address space 1 (partition 2). The IAR resides in address space 0 (partition 1).

Item **6** shows the value of the level status register (LSR). The bits, when set, indicate the following:

- Bits 0–4 — The status of arithmetic operations. Refer to the processor description manual for the meanings of these bits.
- Bit 8 — Program is in supervisor state.
- Bit 9 — Priority level is in process.
- Bit 10 — Class interrupt tracing is active.
- Bit 11 — Interrupt processing is allowed.

Bits 5–7 and bits 12–15 are not used and are always zero.

The LSR value (X'00D0') for level 1 in the sample dump indicates that bits 8, 9, and 11 are set.

Item **7** shows the contents of general-purpose registers R0 through R7 for each hardware interrupt level.

For programs written in EDL, the contents of these registers are described as follows. If the program were written in a language other than EDL, refer to the user's guide for that language to determine the register usage.

R0 Because the supervisor uses this register as a work register, the contents are usually not significant to the failing program.

R1 The address in storage of the failing EDL instruction.

R2 The address in storage of the active task control block (TCB).

R3 The address in storage of EDL operand 1 of the failing instruction.

R4 The address in storage of EDL operand 2 (if applicable) of the failing instruction.

R5 The EDL operation code of the failing instruction. The first byte contains flag bits which indicate how operands are coded. For example, the flag bits indicate whether the operand is in #1, #2, or specified as a constant. The second byte is the operation code of the EDL instruction.

R6 Because the supervisor uses this register as a work register, the contents are usually not significant to the failing program. However, you can determine if the system was emulating EDL code when the failure occurred if R6 is twice the value shown in the second byte of R5. For example, if the second byte of R5 contained X'32' *and* the system was emulating EDL, R6 would contain X'0064'.

R7 Because the supervisor uses this register as a work register, the contents are usually not significant to the failing program.

If the hardware registers in your dump do not follow the EDL register conventions previously discussed, you should examine the IAR and the AKR.

The IAR contains the address of the last machine instruction the system executed when the failure occurred. The AKR tells you in which address space the IAR resides.

To determine where the program failed, you must check the AKR for the correct address space (partition) and check the IAR to find out what was executing at that address.

Look in the supervisor link map from system generation and see if the IAR address is within one of the supervisor modules. If that IAR address appears in the link map, the name of the module that contains the IAR address may give you a clue as to what function was executing when the failure occurred.

Since register usage can vary from one supervisor module to another, the contents of each register may or may not be meaningful to you. You should, however, check the contents of each register.

Sometimes a register may point to a control block. For example, if R3 points to a terminal control block (CCB), you can assume that the program was doing terminal I/O when the failure occurred.

Sometimes the supervisor uses a register (R7 in many cases) for a branch and link instruction. The address in R7 may give you a clue as to which function passed control to the current IAR address.

If the address shown in the IAR is within your program, subtract the program load point from the IAR. Using the resulting address, look in the compiler listing and/or link-edit listing of that program and determine which instruction is at that address and why it failed.

Interpreting the Dump *(continued)*

Floating-Point Registers and Exception Information

Figure 10 shows the next part of the sample dump.

```
8 FR0    FFDF FFFF  FFFF FFFF  FFFF FFFF  FFFF FFFF
         FFFF FFFF  0000 0000  0000 0000  FFFF FFFF

  FR1    FFFF FFFF  FFFF FFDF  0000 0010  0000 0000
         0000 0080  0000 0000  0000 0008  0000 0000

  FR2    00DD FFFF  FFFF FFFF  FFFF FFFF  FFFF FFFE
         FFFF FFFF  0000 0000  0000 0000  FFFF FFFF

  FR3    FFFF FFFF  FFFF FFFF  0000 0000  0000 0000
         0020 0000  0000 0000  0000 0008  0080 0000

9 MACHINE/PROGRAM CHECK LOG BUFFER  - LATEST ENTRY PRINTS LAST

   S/EAK TCBA  PSW  SAR  IAR  AKR  LSR   R0   R1   R2   R3   R4   R5   R6   R7
    0100 0120 8006 B437 2AD6 0000 80D0 0064 850A B520 B437 B434 015C 00B8 0000
```

Figure 10. Floating-point Registers and Exception Information

Item **8** shows the contents of the floating-point registers (FR0–FR3) for each hardware level. This information is printed if the system has the floating-point feature installed.

Item **9** shows entries from the system's software trace table, CIRCBUFF (if included during system generation). The system uses the software trace table to record any program and machine-check entries that occurred since the last IPL. The software trace table is described in greater detail in Chapter 8, "Tracing Exception Information" on page PD-107.

The 2-byte S/EAK field indicates a state variable and an error address key.

The state variable (first byte) can be one of the following values:

0 — No interrupt in process

1 — Standard processing (the default value)

2 — Now processing task error exit

3 — Undefined.

The error address key (second byte) is the address key (1 plus this value is the partition number) that was in use when the error occurred.

The SAR (storage address register) field indicates the address in storage last accessed when the failure occurred.

The remaining fields shown in item **9** also appear in a program check message.

Segmentation Registers

Item **10** in Figure 11 shows the next part of the dump which contains the segmentation registers. In this example, the segmentation registers indicate a system with four partitions and no supervisor mapping across partitions. The partitions are 64K each.

The heading ADS0 represents partition 1, ADS1 represents partition 2, and so on, up through ADS7 which represents partition 8.

The leftmost column (BLOCK) shows the addresses mapped for each segmentation register. Each segmentation register maps 2K of storage. The segmentation registers are listed below each address space (ADS) heading.

10 STORAGE SEGMENTATION REGISTERS:

BLOCK	ADS0	ADS1	ADS2	ADS3	ADS4	ADS5	ADS6	ADS7
0000	0004	0104	0204	0304				
0800	000C	010C	020C	030C				
1000	0014	0114	0214	0314				
1800	001C	011C	021C	031C				
2000	0024	0124	0224	0324				
2800	002C	012C	022C	032C				
3000	0034	0134	0234	0334				
3800	003C	013C	023C	033C				
4000	0044	0144	0244	0344				
4800	004C	014C	024C	034C				
5000	0054	0154	0254	0354				
5800	005C	015C	025C	035C				
6000	0064	0164	0264	0364				
6800	006C	016C	026C	036C				
7000	0074	0174	0274	0374				
7800	007C	017C	027C	037C				
8000	0084	0184	0284	0384				
8800	008C	018C	028C	038C				
9000	0094	0194	0294	0394				
9800	009C	019C	029C	039C				
A000	00A4	01A4	02A4	03A4				
A800	00AC	01AC	02AC	03AC				
B000	00B4	01B4	02B4	03B4				
B800	00BC	01BC	02BC	03BC				
C000	00C4	01C4	02C4	03C4				
C800	00CC	01CC	02CC	03CC				
D000	00D4	01D4	02D4	03D4				
D800	00DC	01DC	02DC	03DC				
E000	00E4	01E4	02E4	03E4				
E800	00EC	01EC	02EC	03EC				
F000	00F4	01F4	02F4	03F4				
F800	00FC	01FC	02FC	03FC				

Figure 11. Segmentation Registers of a Four-partition System

Figure 12 shows another example of the segmentation registers in which the supervisor is mapped across three partitions.

EDX maps partitions starting at address X'0000'. As shown in Figure 12 , address spaces 0 and 1 both have 32 segmentation registers mapped. Address space 2 contains only 10 segmentation registers.

Because the first five segmentation registers in each partition are identical (up to item **1** in Figure 12), you can see that the first 10K of the supervisor in partition 1 is mapped across each partition. Mapping the partitions in this manner leaves partitions 1 and 2 with 54K of storage and partition 3 with 10K of storage which can be used for either supervisor code or application programs.

```
    STORAGE SEGMENTATION REGISTERS:

    BLOCK   ADS0 ADS1 ADS2 ADS3 ADS4 ADS5 ADS6 ADS7

     0000   0004 0004 0004
     0800   000C 000C 000C
     1000   0014 0014 0014
     1800   001C 001C 001C
     2000   0024 0024 0024
1    2800   002C 0104 01DC
     3000   0034 010C 01E4
     3800   003C 0114 01EC
     4000   0044 011C 01F4
     4800   004C 0124 01FC
     5000   0054 012C
     5800   005C 0134
     6000   0064 013C
     6800   006C 0144
     7000   0074 014C
     7800   007C 0154
     8000   0084 015C
     8800   008C 0164
     9000   0094 016C
     9800   009C 0174
     A000   00A4 017C
     A800   00AC 0184
     B000   00B4 018C
     B800   00BC 0194
     C000   00C4 019C
     C800   00CC 01A4
     D000   00D4 01AC
     D800   00DC 01B4
     E000   00E4 01BC
     E800   00EC 01C4
     F000   00F4 01CC
     F800   00FC 01D4
```

Figure 12. Segmentation Registers with Supervisor Mapped Across Partitions

Storage Map

The next section of the sample dump shows the activity in each partition when the dump was taken. This part is called the storage map.

```
    STORAGE MAP:           11 $SYSCOM AT ADDRESS 19C6

12   EDXFLAGS  4000      13 SVCFLAGS   1000

14   PART#   NAME        ADDR PAGES   ATASK   TCB(S)

15    P1      ADS=0      0000   256
16         $TRAP         B400    23 C9E4(A)   C964
           $FSEDIT       CB00    31           E8AC
17         **FREE**      EA00    22

      P2      ADS=1      0000    256          19
18         SAMPLA        0000     4 02C2(A)   0242 01A6 010E 0072
           **FREE**      0400    252

      P3      ADS=2      0000    256
           $SMURON       0000      5           038A
           $DISKUT1      0500     59 2FF6(A)  2F76
           **FREE**      4000    192

20    P4      ADS=3      0000    256
           **FREE**      0000    256
```

Figure 13. Storage Map

Item 11 in Figure 13 shows the address (X'19C6') of the system common area, $SYSCOM (if specified during system generation).

Item 12 is the EDXFLAGS field. The first two digits (40) shown for this field represent the version and modification level of the supervisor. The dump programs do not use the third digit. The last digit (0) indicates the program temporary fix (PTF) level.

Item 13, SVCFLAGS, contains status information. The bits, when set, indicate the following:

- Bit 0 — Supervisor busy
- Bit 1 — Interrupt address (IA) buffer active
- Bit 2 — Dequeue request
- Bit 3 — Floating-point hardware
- Bit 4 — A task is active
- Bit 5 — Remote IPL through Communications Facility
- Bit 6 — WAITM posting in progress
- Bit 7 — Single partition supervisor
- Bit 8 — Supervisor initialization complete
- Bit 9 — Copy of $MEMDISK active
- Bit 10 — Extended Address Mode support active.

Bits 11–15 are not used. The value shown in the example, X'1000', indicates floating-point hardware is installed.

The column headings at item **14** mean the following:

PART# Partition number.

NAME Program name.

ADDR Program load point address.

PAGES The size of the address space (partition) or program in pages. A page is 256 bytes in length. Programs loaded for execution always begin on a page boundary.

ATASK The task control block (TCB) address of the attention list task, if one exists. Task control block addresses of attention list tasks also have **(A)** beside the address.

TCB(S) The task control block addresses in a task chain. The first address in the task chain is always the main task.

Item **15** indicates that partition 1 (address space 0) begins at address X'0000' and is 256 pages in length (64K). Because the whole supervisor resides in partition 1 in this example, the load point of the first program in this partition, $TRAP, begins at address X'B400'. $TRAP is shown at item **16**. The dump also shows that $TRAP is 23 pages in length.

The TCB address X'C9E4' is the address of $TRAP's attention list task. The main TCB for $TRAP is at address X'C964'.

Item **17** indicates the free space in partition 1 beginning at address X'EA00'. The 22 pages of free storage are contiguous.

Item **18** indicates the program SAMPLA is loaded at address X'0000' in partition 2 (address space 1). SAMPLA has an attention list task at address X'02C2'. Also notice that the TCB chain shows the addresses of four task control blocks (item **19**). The task control block at address X'0242' is the main TCB for SAMPLA. The program SAMPLA consists of five task control blocks.

Task control block addresses shown on the TCB chain are the addresses of the tasks defined within the main program. If the main program attaches a task that was link-edited to the main program, and the ATTACH instruction has CHAIN=NO, the address of that task does not appear on the TCB chain.

Because the load point of SAMPLA is at address X'0000', all addresses shown for these tasks would be identical to the compiler listing of SAMPLA.

Item **20** shows that no programs are running in partition 4 (address space 3) and that there are 256 pages of free contiguous storage.

Level Table and TCB Ready Chain

Figure 14 shows the next part of the sample dump.

```
21  EDX LEVEL TABLE - TCB READY CHAIN

    LEVEL ACTIVE       READY (TCB-ADS)


22     1    02C2-1     NONE
23     2    NONE       010E-1 0242-1
       3    NONE       NONE


24  LOADER QCB  CUR-TCB   CHAIN (TCB-ADS)

      94F4  FFFF NONE      NONE
```

Figure 14. Level Table and Task Ready Chain

Item 21 shows the level table and TCB ready chain. The level table keeps pointers to the currently active tasks, all ready tasks for levels 1, 2, and 3, and the address space key in which the tasks reside.

Item 22 shows an active TCB on level 1 at address X'02C2'. The -1 that appears beside this address indicates the address space. Notice also that for level 1, there are no TCBs on the ready chain.

The active TCB at address X'02C2' belongs to the attention list task in partition 2 for program SAMPLA (item 18 in Figure 13 on page PD-80).

Item 23 shows no tasks active on level 2 and two tasks on the ready chain. Notice that these two ready tasks are in address space 1 (partition 2).

The TCB at address X'010E' will be the first task on level 2 to become active if no other task on level 1 or level 2 (with a higher priority) becomes active. Also notice that these two ready tasks reside in program SAMPLA (item 19 in Figure 13 on page PD-80).

Item 24 shows the address (X'94F4') of the loader queue control block (QCB). This address is the entry point of LOADQCB in the resident loader. This entry point appears in the supervisor link map from system generation.

The value X'FFFF' indicates that no tasks are enqueued. If programs were being loaded, this value would be X'0000' and the address of a TCB would be shown.

Terminal Device Information

Figure 15 shows the terminals defined in the supervisor (item 25).

```
25 TERMINAL LIST:

26   NAME     CCB  ID   IODA FEAT QCB CUR-TCB    CHAIN

27 CDRVTA    09FA FFFF 0040 0800 FFFF NONE        NONE
   CDRVTB    0BAA FFFF 0000 0000 FFFF NONE        NONE
28 $SYSLOG   0D84 0406 0004 0400 0000 E8AC-0      NONE
   TERM2     0F5E 040E 0024 0400 0000 02C2-1      NONE
   TERM3     1138 040E 0025 0400 0000 2F76-2      NONE
   $SYSPRTR  131C 0306 0021 0020 FFFF NONE        NONE
   MPRTR     1534 0206 0001 0020 FFFF NONE        NONE
   T3101     177A 2816 0058 0440 FFFF NONE
```

Figure 15. Terminal Device Information

The column headings at item 26 mean the following:

NAME The label on the TERMINAL statement for this device.

CCB The address of the terminal control block (CCB).

ID This value identifies the type of terminal. The values shown also appear when you issue the LD or LS commands of $IOTEST. The value X'FFFF' as shown in item 27 indicates that both CDRVTA and CDRVTB are virtual terminals.

IODA The device address specified on the TERMINAL statement. For virtual terminals, ignore any addresses that appear under this heading.

FEAT This value indicates the device characteristics defined at system generation, such as output pause or spoolable device.

QCB The queue control block (QCB) for the terminal. The value X'FFFF' indicates that no task has enqueued the terminal. If the value were X'0000' as shown in item 28, a task has enqueued the terminal. For example, the task control block at address X'E8AC' in address space 0 (partition 1) belongs to $FSEDIT as shown in the storage map (Figure 13 on page PD-80).

CUR-TCB The address of the task control block and address space of the task currently enqueued on the terminal.

CHAIN The task control block chain. If a task issued an ENQT to any of these terminals while the terminal is currently enqueued by a different task, the TCB address and address space of the task attempting to enqueue that terminal would appear on the chain.

Interpreting the Dump *(continued)*

Disk, Diskette, and Tape Device Information

Information on disk, diskette, and tape devices is presented in Figure 16 , which is the next portion of the dump.

These three device types have volume directory entry (VDE) and device data block (DDB) information listed. The VDE and DDB information is listed under separate headings in the dump. Because of the interrelationship between the VDE and the DDB, the meanings of the headings are explained first.

```
   DISK(ETTE)/TAPE VDE :

29  VDE  NAME   DDB  FLAGS  QCB CUR-TCB    CHAIN (TCB-ADS)

31  06DC *DDE*  0738  0800  FFFF NONE       NONE
    070A EDX002 0738  8000  FFFF NONE       NONE
    07F0 *DDE*  081E  2900  FFFF NONE       NONE

30  DDB  IODA DEVID DSCB-> TASK DSCB-CHAIN

32  0738 0003  00CA 94A6-0 08DE NONE
    081E 0002  0106 CA5A-0 08DE NONE
```

Figure 16. Disk, Diskette, and Tape Device Information

The column headings for the volume directory entry are shown at item 29 and mean the following:

VDE — The volume descriptor entry (VDE) control block describes a volume on disk, diskette, or tape. One VDE is created for each DISK or TAPE statement specified during system generation. If the VOLNAME= operand is coded, one additional VDE is generated for each performance volume.

NAME — The name of the volume. The first VDE for each device is identified as *DDE*. If you coded the VOLNAME= operand on the DISK statement, the performance volumes you specified for the device also appear here.

DDB — The device data block (DDB) describes the physical disk, diskette, or tape device. One DDB is created for each device.

FLAGS — This value indicates information about the volume such as performance volume, diskette, or disk directory.

QCB — The queue control block (QCB) for the disk, diskette, or tape device. The value X'FFFF' indicates that no task has enqueued the device. If the value is X'0000', a task has enqueued the device.

CUR-TCB — The task control block address and address space of the task currently enqueued on the device.

CHAIN — The task control block chain. If a task attempts to enqueue any of these devices while that device is currently enqueued by a different task, the TCB address and address space of the task attempting to enqueue the device would appear on the chain.

The column headings for the device data block (DDB) are shown at item **30** and mean the following:

DDB — The device data block (DDB) describes the physical disk, diskette, or tape device. One DDB is created for each device.

IODA — The device address.

DEVID — The value identifies the type of device. The values shown also appear when you issue the LD or LS commands of $IOTEST.

DSCB-> — A pointer to the data set control block (DSCB) that is currently performing I/O.

TASK — The address of the disk task TCB. If TASK=YES were coded on each DISK or TAPE statement during system generation, one task control block is created for each statement.

DSCB-CHAIN — Identifies the data set control block (DSCB), and its address space, in the chain waiting for service.

If the system encounters erroneous data within a DDB, the dump would show ***ERROR-x** following the line of DDB information. The "x" could be any of the following characters:

A — Control block pointer is an odd address.

D — Address does not exist.

L — Dump facility can dump up to 150 DSCBs. This limit was exceeded.

T — TCB points to itself.

Item **31** in Figure 16 on page PD-84 shows the address of the VDE for a device descriptor entry (DDE). A device descriptor entry describes the entire device and points to the volume directory. The device data block (DDB) for this device is at address X'0738'. Volume EDX002, which was defined as a performance volume, also has X'0738' as the DDB address.

By looking at the DDB address at item **32**, you can obtain further information about this device. This information shows that the device is at address X'0003'. The device ID, X'00CA', means that this device is a 4962 disk model 3.

Because TASK=YES was not specified for either device during system generation, the disk task TCB address (X'08DE') is identical for the DDBs at addresses X'0738' and X'081E'.

Interpreting the Dump *(continued)*

EXIO, BSC, and Timer Information

Figure 17 shows the last part of the formatted control block section of the dump.

```
33 EXIO DEVICE LIST

   NO EXIO DEVICE SYSGENED


34 BSCA DEVICE LIST

   NO BSCA DEVICE SYSGENED


35 7840 TIMER ATTACHMENT

   TIMER DDB   CHAIN (TCB-ADS)                36 10:01:28  mm/dd/yy

37      095E   0072-1 01A6-1
```

Figure 17. EXIO, BSC, and Timer Device Information

Item 33 indicates that no EXIO devices are defined in this system. If any EXIO devices were defined, the DDB address, device type, and device address would appear.

Item 34 also indicates that no binary synchronous communications (BSC) devices are defined. An example of the information you would see if BSC devices were defined follows:

```
BSCA DEVICE LIST

 DDB   ID  IODA

 2864 1006 0009
```

This example shows the DDB at address X'2864'. The value X'1006' indicates a single-line ACCA connection. The device address is X'0009'.

Item 35 indicates the type of timer attached to the system.

Item 36 indicates the time and date of the dump.

Item 37 shows the timer DDB and the TCB address and address space in the TCB chain. If any tasks were executing an STIMER instruction, the entries on the chain are indicated. In this example, the TCBs at addresses X'0072' and X'01A6' (both in address space 1) are on the timer chain. By looking at the storage map section of this sample dump (Figure 13 on page PD-80), you can see that at item 19, these two TCB addresses are on the TCB chain for the program SAMPLA.

Interpreting the Dump *(continued)*

Storage Partition Information

The next portion of the dump shows some of the information dumped from a partition.

```
38 P2    BEGINNING AT ADDRESS 0000 FOR 256 PAGES

39 SNAP DUMP REQUESTED FOR 0000 THRU 045E
                    41                                        42
40  0000    0808 E2C1 D4D7 D3C1 4040 0000 0242 0034    |..SAMPLA  ......|
    0010    0000 0F5E 0344 0000 0000 0000 0100 0342    |...;............|
    0020    0000 0000 0000 02C2 0000 0000 C5C4 E7F0    |.......B....EDX0|
    0030    F0F2 0000 0001 0404 C6C9 D5C9 003E 0019    |02......FINI....|
    0040    004E FFFF 805C 004D 0001 001D 0000 FFFF    |.+...*.(........|
    0050    0000 0001 90A9 1388 0015 0072 FFFF 0015    |................|
     •
     •
     •
    03F0    0000 0000 0000 0000 0000 0000 0000 0000    |................|
43      SAME AS ABOVE
44  0450    0000 0000 0000 0000 0000 0000 0000 0000    |................|
```

Figure 18. Sample Contents of a Partition

Item 38 indicates which partition number was dumped and the size of that partition in pages. In this example, partition 2 was dumped and is 256 pages in length (64K).

Item 39 shows the range of storage addresses dumped. The partition addresses X'0000' through X'0400' appear because the "partial dump" option of $DUMP was selected.

Item 40 shows the beginning address (X'0000') of partition 2. Each line of information shown for an address is 8 words in length. The information shown is the contents of this location in storage when the dump was taken.

Below item 41, the value X'E2C1' is shown. The dump shows that this value is at address X'0002' and begins on a word boundary.

Below item 42 is the EBCDIC representation of the values that were in storage. Thus, the value X'E2C1' shown for item 41 translates to EBCDIC as the characters **SA**. These are the first two characters as shown in the name **SAMPLA**. All characters that are not printable are shown as periods.

The text at item 43 appears in the dump whenever the address that would have been printed for this line contains all null characters (X'00'). In this example, you can see this because the address after X'03F0' is X'0450'.

Item 44 shows the ending address that was specified for the partial dump display.

Interpreting the Dump *(continued)*

Unmapped Storage Information

If you choose to dump and format the unmapped storage areas of your system, the storage dump also will contain a list of unmapped storage pointers. Each unmapped storage pointer refers to a 2K-byte block of unmapped storage that has been obtained by an application program. Figure 19 shows a portion of a dump that contains unmapped storage pointers.

```
1  UNMAPPED STORAGE POINTER -  0780
                2
   0000    AAAA 0000 0000 0000 0000 0000 0000 0000    |................|
   0010    0000 0000 0000 0000 0000 0000 0000 0000    |................|
           SAME AS ABOVE
   0100    AAAA 0000 0000 0000 0000 0000 0000 0000    |................|
   0110    0000 0000 0000 0000 0000 0000 0000 0000    |................|
           SAME AS ABOVE
   •
   •
   •
   0700    AAAA 0000 0000 0000 0000 0000 0000 0000    |................|
   0710    0000 0000 0000 0000 0000 0000 0000 0000    |................|

3  UNMAPPED STORAGE POINTER -  0788

   UNMAPPED STORAGE POINTER -  0790

   UNMAPPED STORAGE POINTER -  0798
```

Figure 19. Unmapped Storage Pointers

Item 1 in the figure shows how the unmapped storage pointers are displayed in the dump.

Beneath item 2 is a listing of the contents of the unmapped storage area. The contents of the unmapped storage area appear in the dump if the area was moved into mapped storage with a SWAP instruction. If, at the time the dump was taken, a program had acquired an area of unmapped storage but had not yet used it, only the pointer to that area appears in the dump. The contents of the area are not listed. See item 3.

When the dump occurs, you may have several programs running which are using unmapped storage. Determining which unmapped storage areas belong to a particular program and which of those areas were in use when the dump occurred is described next.

Locating the Unmapped Storage Areas That Belong to a Program

You can locate the unmapped storage areas that belong to your program by following the steps in this section. You will need the compiler listing for your program and a storage dump. The dump should include the formatted control blocks, a list of the unmapped storage pointers, and the contents of the partition in which the program was running.

The examples in this section refer to a sample program, MAILSORT, and portions of a sample dump.

To identify which unmapped storage areas belong to your program, do the following:

1. Look in the compiler listing for your program and find the address of the STORBLK statement. The STORBLK statement creates a storage control block that defines the size and number of unmapped storage areas your program can use. If your program contains more than one STORBLK statement, repeat the steps described in this section for each statement.

In the sample program, the STORBLK statement is at address X'0034':

```
•
•
•
0034    0000 C1C1 0000 0000 0040  BLK      STORBLK   TWOKBLK=8,MAX=3
003E    0003 FFFF 0000 0000 006A
0048    0000 0000 0000 0000 0000
0066    0000 0000 FFFF FFFF FFFF
0070    FFFF FFFF FFFF FFFF FFFF
```

2. Look at the storage map section of the dump to find the partition in which the program was running and the load point of the program.

In the sample storage map shown below, MAILSORT is in partition 6 (item **1**). The load point of the program is X'5000' (item **2**).

```
STORAGE MAP:          $SYSCOM AT ADDRESS 19C6

EDXFLAGS  4000       SVCFLAGS  1000

PART#  NAME         ADDR PAGES  ATASK  TCB(S)

 P1      ADS=0      0000  256
       **FREE**     0000  256

 P2      ADS=1      0000  256
       **FREE**     3D00    3
       **DATA**     4000   64
       **FREE**     8000  128

 P3      ADS=2      0000  256
       CATALOG      0000    4          0242 01A6 010E
       **FREE**     0400  252

 P4      ADS=3      0000  256
       **FREE**     0000  256

 P5      ADS=4      0000   96
       REORDER      0000    4          0240
       **FREE**     0400   92

 P6      ADS=5      0000  160
       **DATA**     0000   80
                    [2]
[1]    MAILSORT     5000    4          52D0
       **FREE**     5400   76
```

Interpreting the Dump *(continued)*

3. Add the address of the STORBLK statement to the program's load point. The result is the address in storage of the storage control block.

Adding the address of the STORBLK statement in MAILSORT to the program's load point yields a result of X'5034'.

4. Look at the portion of the dump which lists the contents of the partition in which the program was running. Search this portion of the dump for the address you calculated in step 3.

Figure 20 shows the portion of the sample dump that contains the storage control block for MAILSORT. The beginning of the control block is shown at item **1**.

```
P6    BEGINNING AT ADDRESS 0000 FOR 160 PAGES
 •
 •
 •                    [1]
 5030     E2E3 0000 0000 C1C1 0000 0000 0040 0003      |ST....AA..... ..|
                           [2]
 5040     4000 0001 0000 506A 0000 0144 0000 014C      | .....&........<|
 5050     0000 0154 0000 015C 0000 0164 0000 016C      |.......*.......%|
                                     [3]
 5060     0000 0174 0000 017C 0000 078C 0000 0794      |................|
 5070     0000 079C 0000 07A4 0000 07AC 0000 07B4      |................|
 5080     0000 07BC 0000 07C4 0000 07CC 0000 07D4      |.......D.......M|
 5090     0000 07DC 0000 07E4 0000 07EC 0000 07F4      |.......U.......4|
 50A0     0000 07FC 0000 0804 0000 080C 0000 0814      |................|
 50B0     0000 081C 0000 0824 0000 082C 0000 0834      |................|
                           [4]
 50C0     0000 083C 0000 0844 0000 0007 0003 0001      |................|
 50D0     D02A 0001 0000 802C 50CE 50E0 0001 3231      |........&.&/....|
```

Figure 20. Sample Storage Control Block Listing

5. Within the storage control block, find the address of the first pointer to the unmapped storage areas your program obtained. To find this address, do the following:

a. Refer to the list of unmapped storage equates in your program. These equates are generated when you code

```
COPY     STOREQU
```

in your program. The list of equates in the MAILSORT program is as follows:

```
$STRPCHN EQU    0
$STRPID  EQU    $STRPCHN+2
$STRPLEN EQU    $STRPID+2
$STRPRES EQU    $STRPLEN+2
$STORBLK EQU    $STRPRES+2
$STORMAX EQU    $STORBLK+2
$STORMAP EQU    $STORMAX+2
$STORMPK EQU    $STORMAP+2
$STORRSV EQU    $STORMPK+2
$STORUSR EQU    $STORRSV+2
$STORFLG EQU    $STORUSR+2
$STOROVY EQU    X'8000'
$STORMSR EQU    $STORFLG+2
```

Note: The equates shown above are **only** for use in this example. For a current listing of the STOREQU equates, refer to the list generated in your program or refer to the control block equates shown in the *Internal Design*.

b. Find the $STORUSR equate in the list. This equate points to the word in storage that contains the address of the first unmapped storage pointer. The location of $STORUSR in the list reflects the displacement of this word from the beginning of the storage control block.

In the equates in the MAILSORT program, $STORUSR is the tenth equate in the list. Therefore, in this example, the word that contains the address of the first unmapped storage pointer is the tenth word from the beginning of the storage control block. See item **2** in Figure 20 on page PD-90.

c. Using the displacement into the storage control block, find the word that contains the address of the first unmapped storage pointer. When you have found the address of the pointer, locate this address in the dump.

The tenth word from the beginning of the MAILSORT control block contains the address X'506A'. Item **3** in Figure 20 on page PD-90 shows the location of this address in storage. In this example, the first pointer to an unmapped storage area is the doubleword **078C 0000**.

6. Now that you have found the first unmapped storage pointer, refer back to the STORBLK statement in your program. The statement tells you how many 2K-byte blocks of unmapped storage the program obtained. The dump will contain one pointer for each 2K-byte block of unmapped storage. Use the STORBLK statement to calculate the number of unmapped storage pointers your program required.

The STORBLK statement for the MAILSORT program is:

```
BLK    STORBLK    TWOKBLK=8,MAX=3
```

The STORBLK statement defines three unmapped storage areas of 16K-bytes apiece. The number of unmapped storage pointers required then is 24.

7. Note that each unmapped storage pointer is a doubleword. The second word of the doubleword consists of zeros (0000) and can be ignored. Return to the storage dump and, beginning with the first unmapped storage pointer, list the first word of each pointer that belongs to the program.

In the MAILSORT storage control block, the first word of the second unmapped storage pointer is 0794. The first word of third pointer is 079C. The first word of the fourth pointer is 07A4, and so on. The first word of the last pointer (number 24) is 0844. (See item **4** in Figure 20 on page PD-90.)

8. The list of pointer values you collected in step 7 tells you which unmapped storage areas belong to your program. To determine which unmapped storage areas were in use when the dump occurred, look at the portion of the dump that lists the segmentation registers for your system. Scan this list for any of the pointer values that belong to your program. If your program was using a block of unmapped storage when the dump occurred, the pointers to that block of unmapped storage will appear in the segmentation register list.

The following is the list of pointer values which belong to MAILSORT. Each of these values is the first word of an unmapped storage pointer contained in Figure 20.

078C	07CC	080C
0794	07D4	0814
079C	07DC	081C
07A4	07E4	0824
07AC	07EC	082C
07B4	07F4	0834
07BC	07FC	083C
07C4	0804	0844

Figure 21 on page PD-93 shows the segmentation register information in the sample dump. Looking at the segmentation register values for partition 2 (address space 1), you can see eight of the pointer values that belong to MAILSORT. The pointers are highlighted.

The pointers indicate that MAILSORT was using one of the three 16K-byte blocks of unmapped storage it obtained with a GETSTG instruction. The segmentation register values also show that MAILSORT obtained its mapped storage area in partition 2.

STORAGE SEGMENTATION REGISTERS:

BLOCK	ADS0	ADS1	ADS2	ADS3	ADS4	ADS5	ADS6	ADS7
0000	0004	0104	0204	0304	0404	073C		
0800	000C	010C	020C	030C	040C	0744		
1000	0014	0114	0214	0314	0414	074C		
1800	001C	011C	021C	031C	041C	0754		
2000	0024	0124	0224	0324	0424	075C		
2800	002C	012C	022C	032C	042C	0764		
3000	0034	0134	0234	0334	0434	076C		
3800	003C	013C	023C	033C	043C	0774		
4000	0044	080C	0244	0344	0444	077C		
4800	004C	0814	024C	034C	044C	0784		
5000	0054	081C	0254	0354	0454	0464		
5800	005C	0824	025C	035C	045C	046C		
6000	0064	082C	0264	0364	0000	0474		
6800	006C	0834	026C	036C	0000	047C		
7000	0074	083C	0274	0374	0000	0484		
7800	007C	0844	027C	037C	0000	048C		
8000	0084	0184	0284	0384	0000	0494		
8800	008C	018C	028C	038C	0000	049C		
9000	0094	0194	0294	0394	0000	04A4		
9800	009C	019C	029C	039C	0000	04AC		
A000	00A4	01A4	02A4	03A4	0000	0000		
A800	00AC	01AC	02AC	03AC	0000	0000		
B000	00B4	01B4	02B4	03B4	0000	0000		
B800	00BC	01BC	02BC	03BC	0000	0000		
C000	00C4	01C4	02C4	03C4	0000	0000		
C800	00CC	01CC	02CC	03CC	0000	0000		
D000	00D4	01D4	02D4	03D4	0000	0000		
D800	00DC	01DC	02DC	03DC	0000	0000		
E000	00E4	01E4	02E4	03E4	0000	0000		
E800	00EC	01EC	02EC	03EC	0000	0000		
F000	00F4	01F4	02F4	03F4	0000	0000		
F800	00FC	01FC	02FC	03FC	0000	0000		

Figure 21. Sample Segmentation Register Values

Once you find which unmapped storage areas were in use by your program, return to the portion of the dump which lists the contents of the unmapped storage areas (see Figure 19 on page PD-88). You can then examine the contents of the unmapped storage areas that belong to your program.

Analyzing a Failure Using a Storage Dump

Analyzing a Wait State

This section explains how you analyze a wait state using a stand-alone or $TRAP dump. A sample program and portions of a $TRAP dump are presented to show how you analyze the failure.

When you begin analyzing the dump for a wait state, first check to see if a value is shown for the processor status word (PSW). If a value is shown, examine that value to determine if a program check occurred also. The section "How to Interpret the Processor Status Word" on page PD-47 explains what the PSW indicates. If the PSW value does indicate a program check, refer to the section "Analyzing a Program Check" on page PD-100 to help you analyze the failure.

The sample program, WTPGM, prints a test pattern on $SYSPRTR. An ATTNLIST defined in the program *should* enable you to print the test pattern again when you press the attention key and enter **YES**. However, when you attempt to repeat the test pattern, the program enters a wait state.

The following discussion explains how to use the dump and the compiler listing to identify the problem:

1. Look in the storage map section of the dump and find all the task control block (TCB) addresses of the waiting tasks.

As shown for item **1** in the following sample dump, the TCB addresses of the waiting tasks are X'CC28' and X'CBA8'. The task control block at address X'CC28' is the TCB address of the program's attention list task. The task control block at address X'CBA8' is the TCB address of the main task WTPGM.

Notice also for item **2** that the level table shows no active or ready tasks on any hardware level. This further indicates that WTPGM is in a wait state. The dump also shows that $TRAP is not active on any hardware level because the dump was taken using the "programmer console interrupt" option of $TRAP.

```
      STORAGE MAP:          $SYSCOM AT ADDRESS  19C6

      EDXFLAGS  4000        SVCFLAGS  0000

      PART#  NAME         ADDR PAGES  ATASK   TCB(S)

       P1      ADS=0      0000  256
           $TRAP          B400   23 C9E4(A)   C964
 [1]       WTPGM          CB00    2 CC28(A)   CBA8
           **FREE**       CD00   51

       P2      ADS=1      0000   256
           **FREE**       0000   256

       P3      ADS=2      0000   256
           **FREE**       0000   256

       P4      ADS=3      0000   256
           **FREE**       0000   256

      EDX LEVEL TABLE - TCB READY CHAIN

      LEVEL ACTIVE        READY (TCB-ADS)

 [2]    1    NONE         NONE
        2    NONE         NONE
        3    NONE         NONE

      LOADER QCB  CUR-TCB    CHAIN (TCB-ADS)

       94F4  FFFF NONE       NONE
```

Figure 22. Sample Storage Map for a Wait State

Because no tasks were active on any hardware level (except the supervisor on level zero), the section of the dump showing the hardware registers *does not* point to the last instruction executed (R1).

Analyzing a Wait State *(continued)*

```
EVENT DRIVEN EXECUTIVE $TRAP FORMAT STORAGE DUMP

AT TIME OF TRAP PSW WAS 0002 ON HARDWARE LEVEL 0

        LEVEL 0    LEVEL 1    LEVEL 2    LEVEL 3      SVC-LSB   SVCI-LSB
IAR        1F32       1F32       1F32       1F32         1F32       1F0A
AKR        0000       0000       0000       0000         0000       0000
LSR        00C0       0090       0090       0090         00C0       00C0
R0         0000       0000       0000       0000         0000       0000
R1         0000       0000       0000       0000         0000       0000
R2         0000       0000       0000       0000         0000       0000
R3         0000       0000       0000       0000         0000       0000
R4         0000       0000       0000       0000         0000       0000
R5         0000       0001       0002       0003         0000       0002
R6         8000       8000       8000       8000         8000       0000
R7         0000       0000       0000       0000         0000       114C
```

Because you need the address to which R1 is pointing to determine that last instruction executed by each task, you must examine a dump of the partition containing the TCB address for each task. By reviewing the dump of that partition, you can find the address that R1 points to within the TCB of each task.

Figure 23 on page PD-97 shows a sample dump of partition 1. The dump begins at the program's load point (X'CB00') and continues up to the beginning of the free storage area (X'CD00').

2. Do the following to find R1 in the TCB:

a. Look in the dump and find the TCB address (as shown in Figure 22 on page PD-95) of the first task. The first TCB address of the sample program is at address X'CC28'. This address appears under item **1** in Figure 23.

b. Using the TCB equates, find the R1 save area ($TCBS1) in the dump. You locate this field by adding the offset X'0E' to the address of the TCB. In this case, the address X'CC36' points to the address of R1 for the program's attention list task. This address is X'CB60' and appears under item **2**.

c. Subtract the program load point from the address shown for R1. The program load point of the sample program is at X'CB00'. The resulting address for the program's attention list task is X'0060'. You use this address and the compiler listing to identify which instruction the program was executing when the dump was taken. The compiler listing for the sample program is shown in Figure 24 on page PD-98 .

Because the sample program consists of two tasks (an attention list task and the main program), you must also determine what address R1 points to for the second task (main program). The steps you follow are the same as steps 1 through 2c but using the TCB address X'CBA8' of the main task.

The TCB address for the main task is shown under item **3**. The address R1 points to for the main task is X'CB96' and is shown under item **4**.

Again, after subtracting the program load point from the address R1 points to for the main task, the resulting address is X'0096'.

```
P1   BEGINNING AT ADDRESS 0000 FOR 256 PAGES

SNAP DUMP REQUESTED FOR CB00 THRU CD00

 CB00    0808 E6E3 D7C7 D440 4040 0000 CBA8 CB3C    |..WTPGM   ......|
 CB10    0000 0D84 CCAA 0000 0000 0000 0100 CCA8    |................|
 CB20    0000 0000 0000 CC28 CB00 0000 C5C4 E7F0    |............EDX0|
 CB30    F0F2 0000 0000 CBA8 0000 0001 0002 0202    |02..............|
 CB40    D5D6 CB4C 0403 E8C5 E240 CB5A 805C CB3A    |NO.<..YES .!.*..|
 CB50    0002 0019 CB34 FFFF 001D 805C CB3A 0001    |............*....|
 CB60    001D A025 8026 1212 C1C2 C3C4 C5C6 C7C8    |........ABCDEFGH|
 CB70    C9D1 D2D3 D4D5 D6D7 D8D9 8026 1413 E2E3    |IJKLMNOPQR....ST|
 CB80    E4E5 E6E7 E8E9 F1F2 F3F4 F5F6 F7F8 F9F0    |UVWXYZ1234567890|
 CB90    7C40 001A CB34 0017 CB34 A0A2 CB3A 0001    |. ..............|
                              3
 CBA0    CB62 00B2 0022 FFFF FFFF 0000 0000 2098    |................|
                         4
 CBB0    0000 88D0 0000 CB96 CBA8 CB34 A0A2 0017    |................|
 CBC0    002E 2094 0000 02BE 0096 0000 0000 0000    |................|
 CBD0    0000 0000 CBD4 0000 0000 CBD6 C4C5 C2E4    |.....M.....ODEBU|
 CBE0    C740 4040 0000 0000 0000 0000 0000 0000    |G   ............|
 CBF0    0000 FFFF 0000 0000 131C CB00 0000 CBA8    |................|
 CC00    0000 0000 0000 0000 0000 0000 0000 0000    |................|
     SAME AS ABOVE            1
 CC20    0000 0000 CBA8 0080 FFFF 0000 0000 49D6    |...............O|
                         2
 CC30    0000 88D0 0000 CB60 CC28 0D84 FB00 001D    |.......-........|
 CC40    003A 49D2 0000 0001 000A 0000 0000 FFFF    |...K............|
 CC50    0000 0000 CC54 CC28 0D84 CC56 5BC1 E3E3    |............$ATT|
 CC60    C1E2 D240 0000 8000 49CE 0000 0000 0000    |ASK ............|
 CC70    0000 FFFF 0000 0000 0D84 CB00 0000 CBA8    |................|
 CC80    0000 0000 0000 0000 0000 0000 0000 0000    |................|
     SAME AS ABOVE
 CCA0    0000 0000 CC28 0080 0000 0000 0000 0000    |................|
 CCB0    0000 0000 0000 0000 0000 0000 0000 0000    |................|
 CCC0    0000 0000 0000 0000 0000 0000 01CC 0000    |................|
 CCD0    0000 01CE E3C1 E2D2 F340 4040 0000 0000    |....TASK3    ....|
 CCE0    0000 0000 0000 0000 0000 FFFF 0000 0000    |................|
 CCF0    0000 0000 0000 0108 0000 0000 0000 0000    |................|
 CD00    D11E 0000 D11C B0A2 D11E 0000 CD1A 805C    |J...J...J......*|
```

Figure 23. Sample Storage Dump for a Wait State

3. Using the resulting address from step 2c on page PD-96, look at the instruction at that address in the compiler listing and try to determine what caused the wait.

Figure 24 on page PD-98 shows the compiler listing of the sample program. The attention list task points to an ENDATTN instruction at address X'0060'. This address is shown as item **1** in Figure 24 .

Analyzing a Wait State *(continued)*

The main task points to a WAIT instruction at address X'0096'. This address is shown as item **2**.

```
  LOC     +0   +2   +4   +6   +8

  0000    0008 D7D9 D6C7 D9C1 D440  DEBUG   PROGRAM    START
  000A    0000 00A8 003C 0000 0000
  0014    01AA 0000 0000 0000 0100
  001E    01A8 0000 0000 0000 0128
  0028    0000 0000 0000 0000 0000
  0032    0000
  0034    FFFF 0000 0000            EVENT   ECB
  003A    0000                      PRINT   DATA       F'0'
  003C    0002 0202 D5D6 004C 0403  ALIST   ATTNLIST   (NO,POST1,YES,POST2)
  0046    E8C5 E240 005A
  004C                              POST1   EQU        *
  004C    805C 003A 0002                    MOVE       PRINT,2
  0052    0019 0034 FFFF                    POST       EVENT
  0058    001D                              ENDATTN
  005A                              POST2   EQU        *
  005A    805C 003A 0001                    MOVE       PRINT,1
1 0060    001D                               ENDATTN
  0062                              START   EQU        *
  0062    A025                              ENQT       $SYSPRTR
  0064    8026 1212 C1C2 C3C4 C5C6          PRINTEXT   'ABCDEFGHIJKLMNOPQR'
  006E    C7C8 C9D1 D2D3 D4D5 D6D7
  0078    D8D9
  007A    8026 1413 E2E3 E4E5 E6E7          PRINTEXT   'STUVWXYZ1234567890@'
  0084    E8E9 F1F2 F3F4 F5F6 F7F8
  008E    F9F0 7C40
  0092    001A 0034                         RESET      EVENT
2 0096    0017 0034                          WAIT       EVENT
  009A    A0A2 003A 0001 0062               IF         PRINT,EQ,1,START
  00A2    00B2                              DEQT
  00A4    0022 FFFF                         PROGSTOP
  00A8    0000 0000 0000 0234 0000          ENDPROG
  00B2    00D0 0000 0062 00A8 0000
  00BC    0000 0000 0000 0000 0000
   •
   •
   •
  01BE                                      END
```

Figure 24. Compiler Listing of Wait State Program

Because the dump indicates that the attention list task is at the ENDATTN, you can assume the program did pass control to the code at label POST2. The code at POST2 handles the **YES** response. At this label, a value of 1 is moved to the field PRINT. The main task is supposed to repeat the test pattern (branch to START) when PRINT is equal to 1.

By examining the contents of PRINT in the storage dump, you can see that PRINT does contain a 1. The field PRINT is at address X'CB3A' and is under item **5**:

```
P1   BEGINNING AT ADDRESS 0000 FOR 256 PAGES

SNAP DUMP REQUESTED FOR CB00 THRU CD00

 CB00   0808 E6E3 D7C7 D440 4040 0000 CBA8 CB3C    |..WTPGM   ......|
 CB10   0000 0D84 CCAA 0000 0000 0000 0100 CCA8    |................|
 CB20   0000 0000 0000 CC28 CB00 0000 C5C4 E7F0    |............EDX0|
                                  5
 CB30   F0F2 0000 0000 CBA8 0000 0001 0002 0202    |02..............|
  •
  •
  •
```

However, even though the value of PRINT signals the program to repeat the test pattern, the main task is still in a wait state.

By further examining the code at label POST2, notice that an ENDATTN is coded immediately after the MOVE:

```
                                               •
                                               •
                                               •
 005A                                  POST2   EQU      *
 005A   805C 003A 0001                         MOVE     PRINT,1
 0060   001D                                   ENDATTN
 0062                                  START   EQU      *
                                               •
                                               •
                                               •
 0096   0017 0034                              WAIT     EVENT
 009A   A0A2 003A 0001 0062                    IF       PRINT,EQ,1,START
```

Because the main task is waiting on the event control block EVENT to be posted, you must determine what in the program prevents that event control block from being posted.

Closer examination of the code at label POST2 shows that a POST instruction, required to post the event control block, was omitted. Because the attention list routine that processes the **YES** response never posts EVENT, control never passes to the IF instruction which causes a branch to label START.

In order to correct the problem of the wait state in the sample program, the code at label POST2 should look as follows:

```
POST2   EQU       *
        MOVE      PRINT,1
        POST      EVENT
        ENDATTN
```

Analyzing a Program Check

This section explains how you analyze a program check using a stand-alone or $TRAP dump. A sample program, SAMPLA, and portions of a $TRAP dump are presented to show how you analyze the failure.

The failure discussed in this section occurred while SAMPLA, which has an attention list, was executing in partition 2. $FSEDIT was loaded in partition 1 and was enqueued to $SYSLOG. When an operator entered the attention list command **FINI**, the system stopped processing and the terminal from which SAMPLA was loaded would not respond to the attention key. The operator, in this case, IPLed the system, loaded $TRAP to trap all exception types, and reproduced the situation in which the failure occurred. The failure occurred again and the operator printed the dump using $DUMP. The "format control blocks" option was selected.

To analyze the failure, do the following:

1. Look at the portion of the dump that shows the contents of the hardware registers and see if the processor status word (PSW) indicates a program check. The section "How to Interpret the Processor Status Word" on page PD-47 explains the meaning of the PSW.

Note: If a stand-alone dump was taken, begin with step 2 on page PD-101.

Figure 25 shows a portion of the $TRAP dump which contains the hardware registers when the failure occurred:

```
EVENT DRIVEN EXECUTIVE $TRAP FORMAT STORAGE DUMP

1 AT TIME OF TRAP PSW WAS 8006 ON HARDWARE LEVEL 1

          LEVEL 0     LEVEL 1     LEVEL 2     LEVEL 3       SVC-LSB    SVCI-LSB
 IAR         1FFA        2AD6        1F32        1F32          1F32        1F0A
 AKR         0100        0110        0000        0000          0000        0000
 LSR         8090        00D0        0090        0090          00C0        00C0
 R0          0000        0001        0000        0000          0000        0000
 R1        • 0000        0044        0000        0000          0000        0000
 R2          02C2        02C2        0000        0000          0000        0000
 R3          02B6        004D        0000        0000          0000        0000
 R4          0000        0048        0000        0000          0000        0000
 R5          0001        805C        0002        0003          0001        0000
 R6          0000        00B8        8000        8000          8000        0000
 R7          0000        0000        0000        0000          0000        0000
```

Figure 25. Register Contents from Program Check

Because the PSW value shown at item **1** (X'8006') indicates that a program check did occur on level 1, you must determine which task was active on level 1.

2. Look at the level table portion of the dump and find the active task on the highest level.

Figure 26 shows the portion of the sample dump containing the storage map and level table. Item **2** shows that level 1 has an active TCB at address X'02C2' in address space 1 (partition 2). The storage map shows that this TCB is the attention list task (item **3**) for program SAMPLA. The load point for SAMPLA is X'0000'.

```
    STORAGE MAP:            $SYSCOM AT ADDRESS 19C6

    EDXFLAGS  4000        SVCFLAGS  1000

    PART#  NAME          ADDR PAGES  ATASK   TCB(S)

     P1      ADS=0       0000  256
          $TRAP          B400   23 C9E4(A)   C964
          $FSEDIT        CB00   31           E8AC
          **FREE**       EA00   22

     P2      ADS=1       0000  256 [3]
          SAMPLA         0000    4 02C2(A)   0242 01A6 010E 0072
          **FREE**       0400  252

     P3      ADS=2       0000  256
          **FREE**       0000  256

     P4      ADS=3       0000  256
          **FREE**       0000  256

    EDX LEVEL TABLE - TCB READY CHAIN

    LEVEL ACTIVE         READY (TCB-ADS)

[2]    1     02C2-1      NONE
       2     NONE        010E-1 0242-1
       3     NONE        NONE

   LOADER QCB   CUR-TCB    CHAIN (TCB-ADS)

    94F4   FFFF NONE       NONE
```

Figure 26. Storage Map and Level Table for Program Check

Analyzing a Program Check *(continued)*

3. Look at the portion of the dump containing the hardware registers and see if the address of the active TCB is in R2 of the level 1 registers.

At item **4** in the following example, notice that the address for R2 on level 1 does show the address X'02C2'.

```
EVENT DRIVEN EXECUTIVE $TRAP FORMAT STORAGE DUMP

AT TIME OF TRAP PSW WAS 8006 ON HARDWARE LEVEL 1

         LEVEL 0     LEVEL 1     LEVEL 2     LEVEL 3          SVC-LSB    SVCI-LSB
IAR         1FFA        2AD6        1F32        1F32             1F32        1F0A
AKR         0100        0110        0000        0000             0000        0000
LSR         8090        00D0        0090        0090             00C0        00C0
R0          0000        0001        0000        0000             0000        0000
R1          0000      5 0044        0000        0000             0000        0000
R2          02C2      4 02C2        0000        0000             0000        0000
R3          02B6        004D        0000        0000             0000        0000
R4          0000        0048        0000        0000             0000        0000
R5          0001        805C        0002        0003             0001        0000
R6          0000        00B8        8000        8000             8000        0000
R7          0000        0000        0000        0000             0000        0000
```

Notice also that the address for R1 (item **5**), which points to the failing EDL instruction, points to address X'0044'. Because the program load point for SAMPLA is at address X'0000', the address X'0044' corresponds to address X'0044' in the compiler listing of SAMPLA.

When a program load point is other than X'0000', subtract the load point address from the address of R1. Use the resulting address to find the failing EDL instruction in the compiler listing.

4. Using the address of the failing EDL instruction (the address in R1 in this case), look at that address in the compiler listing and determine the cause of the failure.

Figure 27 shows the compiler listing for the program SAMPLA. As shown for item 6, notice that at address X'0044' the program attempts to move a word of data to an odd-byte boundary (WORD+1).

```
  LOC     +0   +2   +4   +6   +8

                                              PRINT       NODATA
  0000    0008 D7D9 D6C7 D9C1 D440   SAMPLA   PROGRAM     START
  0034    0001 0404 C6C9 D5C9 003E            ATTNLIST    (FINI,DONE)
  003E    0019 004E FFFF             DONE     POST        ECB
6 0044    805C 004D 0001                      MOVE        WORD+1  ,1
  004A    001D                                ENDATTN
  004C    0000                       WORD     DC          F'0'
  004E    0000 0000 0000             ECB      ECB         0
  0054    90A9 1388                  START    STIMER      5000,WAIT
  0058    0015 0072 FFFF                      ATTACH      TASK1
  005E    0015 010E FFFF                      ATTACH      TASK2
  0064    0015 01A6 FFFF                      ATTACH      TASK3
  006A    0017 004E                           WAIT        ECB
  006E    00A0 023E                           GOTO        END
  0072    0000 0000 0000 0234 0000   TASK1    TASK        START1
  00F2    835C 0000 0014             START1   MOVE        #1,20
                                              •
                                              •
                                              •
  0106    0016 FFFF 00A0 00F2                 ENDTASK
  010E    0000 0000 0000 0234 0000   TASK2    TASK        START2
  018E    835C 0000 0028             START2   MOVE        #1,40
                                              •
                                              •
                                              •
  019E    0016 FFFF 00A0 018E                 ENDTASK
  01A6    0000 0000 0000 0234 0000   TASK3    TASK        START3
  0226    835C 0000 0080             START3   MOVE        #1,128
                                              •
                                              •
                                              •
  0236    0016 FFFF 00A0 0226                 ENDTASK
  023E    0022 FFFF                  END      PROGSTOP
  0242    0000 0000 0000 0234 0000            ENDPROG
                                              END
```

Figure 27. Compiler Listing of Program Check Program

Analyzing a Program Check *(continued)*

In the following example of the hardware registers for level 1, item 7 shows that R3 (operand 1) is at address X'004D', which is on an odd-byte boundary. Item 8 shows that the address of R4 (operand 2) is at address X'0048', which is on a word boundary. Thus, any attempt to move a word of data to a byte boundary causes a specification check as indicated by item 1.

```
   EVENT DRIVEN EXECUTIVE $TRAP FORMAT STORAGE DUMP

1  AT TIME OF TRAP PSW WAS 8006 ON HARDWARE LEVEL 1

           LEVEL 0    LEVEL 1    LEVEL 2    LEVEL 3       SVC-LSB   SVCI-LSB
   IAR       1FFA       2AD6       1F32       1F32          1F32       1F0A
   AKR       0100       0110       0000       0000          0000       0000
   LSR       8090       00D0       0090       0090          00C0       00C0
   R0        0000       0001       0000       0000          0000       0000
   R1        0000       0044       0000       0000          0000       0000
   R2        02C2       02C2       0000       0000          0000       0000
   R3        02B6     7 004D       0000       0000          0000       0000
   R4        0000     8 0048       0000       0000          0000       0000
   R5        0001       805C       0002       0003          0001       0000
   R6        0000       00B8       8000       8000          8000       0000
   R7        0000       0000       0000       0000          0000       0000
```

Because $FSEDIT had the $SYSLOG terminal enqueued, the system was unable to display the program check message, and as a result, caused the system to stop processing.

Analyzing a Run Loop

This section explains an approach you can use to analyze a run loop with the help of a stand-alone or $TRAP dump.

Because a run loop occurs within a range of instruction addresses in a program, the dump would only show the instruction address at which the program was executing when the dump was taken. You can, however, use a dump to identify which task was active and the hardware level on which the task was executing.

To analyze a run loop using a dump, do the following:

1. Look at the level table in the dump and find the TCB address of the active task on the highest level.

2. Look in the storage map of the dump and find the name of the program whose TCB address matches the TCB address from step 1 .

3. Rerun that program.

4. Turn to the section "Determining the Starting and Ending Points of the Loop" on page PD-21. That section explains how to trace the addresses within the loop using $DEBUG.

Notes

Chapter 8. Tracing Exception Information

The system sets aside an area in storage that it uses to record program check, soft exception, and machine check information. This area in storage is called the software trace table. However, in order for this storage area to be present, you must include the module CIRCBUFF during system generation.

The software trace table provides you with an alternate method of identifying the cause of an exception. For example, if for some reason you were not able to record the information displayed in a program check message, you could use the information in the trace table to help you analyze the exception.

The system makes an entry into the software trace table when an exception occurs. The system does not record exceptions that occur in a program or task that has the ERRXIT= operand coded on the PROGRAM or TASK statement.

The software trace table can contain a maximum of eight entries. When the maximum number of entires is reached, the system overlays the oldest entry in the table with the newest entry. Thus, the system records these entries in a "circular" fashion.

The entries in the trace table reflect the number of exceptions since the last IPL. The system resets (clears) this table during each IPL.

If any entries are in the trace table when you take a stand-alone or $TRAP dump, these entries are also shown in the dump. Figure 10 on page PD-76 shows an example of how an entry appears in a dump.

You can display the contents of the trace table on a terminal using the $D operator command. How you do this is described next.

Tracing Exception Information

Displaying the Software Trace Table

You can display the contents of the software trace table at your terminal. In order to display the trace table, first you need the supervisor link map listing from system generation.

To display the software trace table, do the following:

1. Change your terminal to partition 1 by pressing the attention key and entering **$CP 1**.

2. Press the attention key and enter **$D**.

3. At the prompt for ORIGIN:, enter **0000**.

The next prompt, ADDRESS,COUNT:, asks you for an address and the number of words you want to display.

4. For ADDRESS, enter the address of the software trace table. The address of the software trace table appears beside the entry point name CIRCBUFF in the supervisor link map listing.

5. For COUNT, enter the value **125**. This value is the number of words in storage the trace table occupies.

The system then displays the contents of the trace table at the terminal. An explanation of the information displayed is in the section "Software Trace Table Format" on page PD-110.

6. Reply **N** to the prompt ANOTHER DISPLAY?

Displaying the Software Trace Table *(continued)*

Figure 28 is an example showing steps 1 through 5 on page PD-108. The address of the trace table (CIRCBUFF) in this example is X'8F64'. The trace table contains two entries.

```
> $CP 1

PROGRAMS AT 00:00:15
IN PARTITION #1  NONE
         PARTITION ADDRESS: B400 HEX;  SIZE:  19456 DECIMAL BYTES
> $D
ENTER ORIGIN: 0000
ENTER ADDRESS,COUNT: 8F64,125
 8F64: 8F6E 8FAA 905E 0002 001E 0100 0120 8002
 8F74: B437 2AD6 0000 80D0 0064 B50A B520 B437
 8F84: B434 015C 00B8 0000 0101 01A8 8002 01A9
 8F94: 2B86 0110 80D0 0192 013C 01A8 019A 01A9
 8FA4: 005E 00BC 0000 0000 0000 0000 0000 0000
 8FB4: 0000 0000 0000 0000 0000 0000 0000 0000
 8FC4: 0000 0000 0000 0000 0000 0000 0000 0000
 8FD4: 0000 0000 0000 0000 0000 0000 0000 0000
 8FE4: 0000 0000 0000 0000 0000 0000 0000 0000
 8FF4: 0000 0000 0000 0000 0000 0000 0000 0000
 9004: 0000 0000 0000 0000 0000 0000 0000 0000
 9014: 0000 0000 0000 0000 0000 0000 0000 0000
 9024: 0000 0000 0000 0000 0000 0000 0000 0000
 9034: 0000 0000 0000 0000 0000 0000 0000 0000
 9044: 0000 0000 0000 0000 0000 0000 0000 0000
 9054: 0000 0000 0000 0000 0000
```

Figure 28. Sample Software Trace Table Entries

The next section explains the format and contents of the software trace table.

Tracing Exception Information

Software Trace Table Format

The software trace table is a 125-word area in processor storage. The trace table consists of control information and exception entries. This area in storage is described in the following sections.

Control Information Format

The first 5 words of the trace table are control information. This 5-word area contains the following information:

Word	*Contents*
0	The address of the first entry in the table.
1	The address at which the next entry will be written.
2	The ending address of the table. This address points to the first byte beyond the end of the table.
3	The number of exceptions that occurred since the last IPL.
4	The size (in bytes) of each entry in the table. This field contains the value X'1E' which indicates each entry is 30 bytes (15 words) in length.

Figure 29 shows several lines of control information from the previous example. An explanation of each numbered item follows the figure.

```
          1    2    3         4    5    6
8F64:  8F6E 8FAA 905E 0002 001E 0100 0120 8002
8F74:  B437 2AD6 0000 80D0 0064 B50A B520 B437
                                7
8F84:  B434 015C 00B8 0000 0101 01A8 8002 01A9
8F94:  2B86 0110 80D0 0192 013C 01A8 019A 01A9
                           8
8FA4:  005E 00BC 0000 0000 0000 0000 0000 0000
 •
 •
 •
 •                              9
9054:  0000 0000 0000 0000 0000
```

Figure 29. Control Information Example

The address (X'8F6E') shown below item **1** points to the first exception entry in the trace table. The first exception entry is shown below item **6**.

The address (X'8FAA') shown below item **2** points to the address at which the next exception entry will be written. This address is shown below item **8**.

Item **3** points to the first byte of storage following the trace table. This address (X'905E') is not shown in the example, but would begin immediately *after* item **9**.

Item **4** indicates that two exceptions have occurred since the last IPL. The second exception entry begins below item **7**.

The value (X'001E') below item **5** indicates the length (in bytes) of each entry.

The next section explains the format and contents of an exception entry.

Software Trace Table Format *(continued)*

Exception Entry Format

Each exception entry in the trace table is 15 words (30 bytes) in length. The *first* entry, which follows the five words of control information, begins at word 5 in the table. When the maximum number of entries (eight) is reached, the system writes the next entry at word 5 again, overlaying the previous entry. Each entry contains the following information:

Word ***Contents***

0 This word contains a state variable and an address key.

The state variable, which is the first byte, can have any of the following values:

0 — No interrupt in process

1 — Standard (default) processing

2 — Now processing task error exit

3 — Undefined

The address key, which is the second byte, indicates the address space that was in use when the exception occurred. The partition in which the exception occurred is this value plus 1.

1 The task control block (TCB) address of the failing task.

2 The value of the processor status word (PSW). The section "How to Interpret the Processor Status Word" on page PD-47 explains the meaning of this value.

3 The contents of the storage address register (SAR). This field indicates the address in storage last accessed when the failure occurred.

4 The contents of the instruction address register (IAR). This field indicates the address of the machine instruction currently executing.

5 The contents of the address key register (AKR). The last 3-hexadecimal digits indicate in which address space operand 1, operand 2, and the IAR reside. Bit 0 of the AKR is the equate operand spaces (EOS) bit. If bit 0 is set to 1, the address space key indicated for operand 2 is the address space key used for operand 1 and operand 2.

6 The contents of the level status register (LSR). The bits, when set, indicate the following:

Bits 0–4 — The status of arithmetic operations. Refer to the processor description manual for the meanings of these bits.

Bit 8 — Program is in supervisor state.

Bit 9 — Priority level is in process.

Bit 10 — Class interrupt tracing is active.

Bit 11 — Interrupt processing is allowed.

Bits 5–7 and bits 12–15 are not used and are always zero.

7 The contents of hardware register 0 (R0). Because the supervisor uses this register as a work register, the contents are usually not significant to the failing program.

8 The contents of hardware register 1 (R1). This field contains the address in storage of the failing EDL instruction.

9 The contents of hardware register 2 (R2). This field contains the address in storage of the active task control block (TCB).

10 The contents of hardware register 3 (R3). This field contains the address in storage of EDL operand 1 of the failing instruction.

11 The contents of hardware register 4 (R4). This field contains the address in storage of EDL operand 2 (if applicable) of the failing instruction.

12 The contents of hardware register 5 (R5). This field contains the EDL operation code of the failing instruction. The first byte contains flag bits which indicate how operands are coded. For example, the flag bits indicate whether the operand is in #1, #2, or specified as a constant. The second byte is the operation code of the EDL instruction.

13 The contents of hardware register 6 (R6). Because the supervisor uses this register as a work register, the contents are usually not significant to the failing program. However, you can determine if the system was emulating EDL code when the failure occurred if R6 is twice the value shown in the second byte of R5. For example, if the second byte of R5 contained X'32' *and* the system was emulating EDL, R6 would contain X'0064'.

14 The contents of hardware register 7 (R7). Because the supervisor uses this register as a work register, the contents are usually not significant to the failing program. However, in many cases, R7 may contain the address of a branch and link instruction. The address may give you a clue as to which module passed control to the address in the IAR.

Software Trace Table Format *(continued)*

Excluding the address of the program load point, all entries in the trace table contain the same information that the system displays in a program check message, *plus* two additional fields: the state variable and address key word, and the storage address register (SAR). The section "Finding the Program Load Point Address" on page PD-115 explains how you can find the address of the program load point.

The following application program check message caused the system to create the exception entry in the trace table shown below the message.

```
PROGRAM CHECK:
 PLP  TCB  PSW  IAR  AKR  LSR  R0   R1   R2   R3   R4   R5   R6   R7
 B400 0120 8002 2AD6 0000 80D0 0064 B50A B520 B437 B434 015C 00B8 0000
```

The exception entry for the previous program check message begins below item **1** and ends below item **15**.

```
                                          1    2    3
8F64:  8F6E 8FAA 905E 0002 001E 0100 0120 8002
         4    5    6    7    8    9   10   11
8F74:  B437 2AD6 0000 80D0 0064 B50A B520 B437
        12   13   14   15
8F84:  B434 015C 00B8 0000 0101 01A8 8002 01A9
```

Item **1** shows the value of the state variable and address key. The value of the state variable (X'01') indicates standard processing. The address key indicates address space 0 (partition 1).

Item **2** shows the task control block (TCB) address X'0120'.

Item **3** shows the value of the processor status word (PSW). The value X'8002' indicates a specification check occurred and that the translator was enabled. The specification check was caused by a word move to a odd-byte boundary.

Item **4** shows the value (X'B437') of the storage address register (SAR).

Item **5** shows the value (X'2AD6') of the instruction address register (IAR).

Item **6** shows the value (X'0000') of the address key register (AKR).

Item **7** shows the value (X'80D0') of the level status register (LSR).

Items **8** through **15** show the contents of hardware registers R0 through R7.

Finding the Program Load Point Address

In order to determine where the failure occurred in the application program, you need the address of the program load point. An exception entry in the trace table does not contain this address, but you can find the load point address by using the value of the address key and the TCB address.

If the area in storage that contained the failing program's task control block (TCB) has been overlaid by other active tasks, you *cannot* find the load point address in the failing program's TCB. The note under step 1 may apply, however.

This discussion assumes that you are using the most recent exception entry in the trace table and that you were unable to record the program check message displayed for this exception. The following steps explain how to find the program load point address:

1. Look at the value in the address key (word 0, second byte) and determine the partition in which the failing program was active.

   Note: If the failing program was the *only* program active in that partition, the load point address is the address at which the partition begins. The $A ALL operator command displays the beginning address of each partition. Using the beginning address of that partition as the program load point address and the rest of the information in the exception entry, turn to the section "How to Analyze an Application Program Check" on page PD-54.

   If multiple programs were active in that partition, go to step 2 .

2. Add the value X'52' to the address shown for the TCB (word 1 in the exception entry). Adding this value to the TCB address points to the field $TCBPLP in the task control block. $TCBPLP contains the program load point address.

3. Press the attention key and enter **$CP** specifying the partition number from step 1.

4. Press the attention key and enter **$D**.

5. At the prompt for ORIGIN:, enter **0000**.

6. At the prompt for ADDRESS,COUNT:, enter the address you calculated in step 2. Enter the value 1 for the count.

The value the system displays is the program load point address of the failing program.

7. Reply **N** to the prompt ANOTHER DISPLAY?

Finding the Program Load Point Address *(continued)*

The following items are ways in which you can determine if the program load point is valid:

- Check to see if the address is within the size of the partition in which the program was running.
- Subtract the load point address from the address shown for R1 (word 8 in the exception entry). Using the resulting address and the compiler listing of the failing program, determine if that address is within the program.
- Make sure that if the address is within the program, it is the address of an executable instruction.

If all of the above items seem correct, the address of the program load point is probably valid and belongs to the failing program. Using this program load point address and the rest of the information in the exception entry, turn to the section "How to Analyze an Application Program Check" on page PD-54.

Chapter 9. Recording Device I/O Errors and Program Check Information

When the system detects an I/O error for a device or encounters an error that interrupts normal processing, it can supply you with information to help you pinpoint the cause of the problem. The $LOG utility provides you with a way to record such error information whenever the system issues it.

Typically, when the system detects an I/O error for one of the devices attached to your Series/1, it issues status information about the device. When $LOG is active, it writes this status information to a log data set on disk or diskette. If the system encounters an error during processing that causes it to issue a program check message, $LOG, if active, also writes the contents of the program check message to a log data set. The system provides two types of program check messages: a system program check and an application program check. (Refer to Chapter 6, "Analyzing and Isolating a Program Check" on page PD-43 for more details on program check messages.)

The information stored in the system's log data set lets you know that an error occurred and can help you to find the source of the error. The information is especially useful when you are experiencing intermittent I/O errors and you have to call a service representative to analyze the problem. For this reason, it is recommended that you keep the $LOG utility active whenever you operate your system.

This chapter explains how to allocate a log data set, how to load and run $LOG, how to print or display the information in the log data set, and how to interpret the error information you receive.

Recording Device I/O Errors and Program Check Information

Allocating the Log Data Set

Before you use $LOG, you must allocate a data set to contain any error information that the utility may record. You can allocate the log data set with the AL command of the $DISKUT1 utility. You can name the data set anything you wish (1 to 8 characters) and it can reside on any disk or diskette volume.

$LOG writes a single 256-byte record in the log data set for each device I/O error or program check error that it records. Allocate as many records for the data set as you feel you require. You must allocate at least three records because $LOG uses the first two records of the data set for control information.

The following example shows how to allocate a log data set that can contain up to 30 "log records." In the example, the name of the log data set is LOGDS and it resides on volume EDX002:

```
> $L $DISKUT1
LOADING $DISKUT1     59P,00:00:15, LP= B400, PART= 1

$DISKUT1 - DATA SET MANAGEMENT UTILITY I

USING VOLUME EDX002

COMMAND (?): AL LOGDS 30 D
LOGDS CREATED

COMMAND (?): EN

$DISKUT1 ENDED AT 00:00:30
```

Figure 30. Example of Allocating a Log Data Set

If the log data set is empty, $LOG begins writing to the third record in the data set. If the log data set already contains entries, $LOG adds new entries after the old ones.

Starting and Controlling Error Logging

To start error logging, use the $L operator command to load $LOG into any partition. $LOG asks you for the name and volume of the log data set. After you specify the data set name and volume, $LOG is ready to start logging errors.

Note: For the remote manager (RM1) to receive error log information, you must also load either the host program (CJUALTHL) or the send program (CJUALTSL).

Figure 31 on page PD-119 shows an example of how to start error logging. In this example, item **1** shows how to load $LOG. The prompt message at item **2** requests the name and volume of the log data set. The example uses the data set created in Figure 30.

Item 3 shows the attention commands you can enter to control $LOG. You can issue those commands at any time. $LOG displays the message shown at item 4 to indicate logging is active.

```
1  > $L $LOG
2   LOGDS   (NAME,VOLUME): LOGDS,EDX002
   LOADING $LOG          23P,00:00:15, LP=0000, PART=2

  *******************************************************
  *       $LOG UTILITY
  *
3 *  THE FOLLOWING ATTENTION COMMANDS ARE AVAILABLE:
  *     ATTN/$LOGOFF    - TEMPORARILY DEACTIVATE LOGGING
  *     ATTN/$LOGON     - REACTIVATE LOGGING
  *     ATTN/$LOGINIT   - INITIALIZE LOG DATA SET
  *                       REACTIVATE LOGGING
  *     ATTN/$LOGTERM   - TERMINATE LOGGING
  *     ATTN/$LOG       - REISSUE COMMAND LIST
  *     ATTN/$LOGDISP   - DISPLAY ERROR MSG ON OCCURRENCE
  *     ATTN/$LOGTDW    - TERMINATE ON DATA SET WRAP
  *
  *   WARNING: DO NOT CANCEL ($C) THIS PROGRAM
  *
  *******************************************************

4  LOGGING ACTIVATED
```

Figure 31. Example of Starting Error Logging

As shown in Figure 31, $LOG has attention commands that enable you to control its activity. To issue a command, press the attention key, type in the command name, and press the enter key. You can use the commands as follows:

Command	***Use***
$LOGOFF	Suspend error logging ($LOG is still loaded).
$LOGON	Restart error logging
$LOGINIT	Clear the log data set and restart error logging. When you use the $LOGINIT command, the system writes a new log control record to indicate that no entries are in the log data set.
$LOGTERM	End error logging ($LOG is no longer loaded).
$LOG	Display the list of attention commands. This command also displays any error messages issued by the $LOG utility. These error messages are from the utility itself and have nothing to do with the errors that $LOG is tracking.

Starting and Controlling Error Logging *(continued)*

Command	*Use*
$LOGDISP	Display any error messages issued **by** $LOG when they occur. For example, if the log data set becomes full during error logging, an error message will be displayed immediately. If you don't enter $LOGDISP, you must use the $LOG command to display errors.
$LOGTDW	End the $LOG utility if the log data set becomes full during error logging. If you do not enter this command, $LOG returns to the third record in the data set and begins writing over the existing entries.

Printing or Displaying the Log Information

By reviewing the log information, you can determine if any device I/O errors or program check errors occurred while $LOG was active. The $DISKUT2 utility enables you to display the log information on a terminal (LL command) or print it on any printer (PL command).

Note: If you use the remote manager (RM1), use the LR command of $DISKUT2 to display the log information on a terminal or the PR command to print the information on a printer. See the *Operator Commands and Utilities Reference* for more information on using these commands.

$DISKUT2 also enables you to display or print the following:

- The log entries for an I/O device at a particular address. If you do not know the I/O device addresses on your system, load the $IOTEST utility and issue the LS or LD command.
- The log entries for all program checks issued by the system while $LOG was active (application program checks and system program checks).
- All log entries in the log data set. This includes log entries for the I/O devices on your system and for program check errors.

Figure 32 shows an example of how to print all of the log entries in the log data set. An explanation of the numbered items follows the example.

```
1  > $L $DISKUT2
   LOADING $DISKUT2        51P,00:29:36, LP=0000, PART= 2

   $DISKUT2 - DATA SET MGMT. UTILITY II

2  USING VOLUME EDX002

3  COMMAND(?): PL
4  LOG DS NAME: LOGDS
5  ENTER DEVICE ADDRESS, NULL FOR ALL ENTRIES,
   OR 'FFFF' FOR PROGRAM/SYSTEM CHECKS ONLY:

6  DUMP ALL OF LOG? Y

   COMMAND(?): EN

   $DISKUT2 ENDED AT 00:30:34
```

Figure 32. Example of Printing the Log Data Set

Item 1 shows how you load $DISKUT2 after pressing the attention key.

As shown at item 2, $DISKUT2 assumes that you are using the IPL volume. If the log data set does not reside on the IPL volume, enter the CV command (change volume) at the first COMMAND prompt and specify the volume on which the log data set resides.

The PL command entered at item 3 instructs the utility to print the log information on a printer. $SYSPRTR is the default printer. If you enter the LL command, $DISKUT2 displays the log information on your terminal.

The prompt message at item 4 asks for the name of the log data set. In this example, the name of the log data set is LOGDS.

The prompt message shown at item 5 asks you to specify the type of log information you want printed. To print the log entries for a specific I/O device, enter the address of the device. To print only the log information on program check errors, enter **FFFF**. Press the enter key if you want to print all of the log entries in the log data set. In this example, the enter key is pressed (a null reply) to print the contents of the entire log data set.

If you press the enter key, you receive the prompt shown at item 6. Reply **Y** to confirm your choice to print the entire log data set. If you reply **N**, $DISKUT2 asks you again for the type of log information you want printed.

Printing or Displaying the Log Information *(continued)*

Interpreting the Printed Output

The figures in this se.. 'es of the printed output created by $DISKUT2. An explanation of the numbered items follows each example. Figure 33 shows the general format of error log entries for I/O devices.

```
1  ERROR LOG LIST, DATASET: LOGDS   ON EDX002
2  I/O LOG ERROR COUNTERS (BY DEVICE ADDR):
                        3
     0000      0000 0100 0000 0000 0000 0000 0000 0000
     0010      0000 0000 0000 0000 0000 0000 0000 0000
                  4
     0020      0001 0000 0000 0000 0000 0000 0000 0000
     0030      0000 0000 0000 0000 0000 0000 0000 0000
     0040      0000 0000 0000 0000 0000 0000 0000 0000
     0050      0000 0000 0000 0000 0000 0000 0000 0000
     0060      0000 0000 0000 0000 0000 0000 0000 0000
     0070      0000 0000 0000 0000 0000 0000 0000 0000
     0080      0000 0000 0000 0000 0000 0000 0000 0000
     0090      0000 0000 0000 0000 0000 0000 0000 0000
     00A0      0000 0000 0000 0000 0000 0000 0000 0000
     00B0      0000 0000 0000 0000 0000 0000 0000 0000
     00C0      0000 0000 0000 0000 0000 0000 0000 0000
     00D0      0000 0000 0000 0000 0000 0000 0000 0000
     00E0      0000 0000 0000 0000 0000 0000 0000 0000
     00F0      0000 0000 0000 0000 0000 0000 0000 0000

5  PERM ERR
   6                   7
   DEV ADDR:  0002     DEV ID:  0106
   8                   9                10
   DATE:  9/15/84      LVL:  0001       AKR:    0000
   11                  12              13
   TIME:  0:20:22      RETRY: 10      IDCB:   7002 0852
   14                  15
   INTCC:  0002        ISB:  0080
16 DCB 1:  8007 0000 0000 0000 0000 0862 0000 0000

   DCB 2:  8005 0001 0000 0001 0000 0872 0000 0000

   DCB 3:  2109 0000 0000 1001 0001 0000 0100 1D4C

17 CSSW:    0881 4000 1001 0001

   PERM ERR        18                  19
   DEV ADDR:  0021     DEV ID:  0306
   DATE:  9/16/84      LVL:  0003     AKR:    0100
   TIME:  0: 2:53      RETRY: 2     IDCB:   0000 0000
   INTCC:  0002        ISB:  0080
   CSSW:    12D1 2041 0015 4200 0000 FFFF 00F8 6080
   LOG LISTING ENDED
```

Figure 33. Example of Log Entries for I/O devices

Printing or Displaying the Log Information *(continued)*

Item 1 identifies the name and volume of the log data set $DISKUT2 is printing. In this example, the log data set is LOGDS on volume EDX002.

The information shown below item 2 lists device addresses and the number of I/O errors that have occurred at those addresses. The device addresses range from X'00'–X'FF', or 0–255.

Each byte indicates a device address and the number of I/O errors (in hexadecimal) logged at that address since the log data set was last initialized. For example, the value X'01' shown below item 3 indicates that one I/O error occurred at device address X'02'. Further, item 4 indicates that one I/O error occurred at device address X'21'.

Item 5 indicates the type of I/O error. $DISKUT2 indicates either a permanent error (PERM ERR) or a soft-recoverable error (SOFT RECOV ERR). A permanent error is an I/O error from which the device cannot recover after attempting to retry the I/O operation.

A soft-recoverable error is one that through retrying the I/O operation, the device is able to recover from the error.

Item 6 identifies the address of the device encountering the I/O error. The device address is contained in the right-most byte of the word. In this example, the device is at address X'02'

Item 7 identifies the device type. The value X'0106' in this the example, indicates a 4964 diskette unit. The device type is also shown when you issue the LS or LD command of $IOTEST.

Item 8 shows the date, according to the system clock, when the I/O error occurred.

Item 9 indicates the the hardware interrupt level that was active when the I/O error occurred. This example shows that hardware interrupt level 1 was active.

Item 10 shows the value of the address key register (AKR). This value indicates the address space that contained the active task when the error occurred. In this example, address space 0 (partition 1) contained the active task.

Item 11 shows the time, according to the system clock, when the I/O error occurred.

Item 12 shows the number of times that the supervisor issued the I/O instruction to the device before logging the error.

Item 13 shows two words of immediate device control block (IDCB) information. The first word contains the I/O operation and the device address. The second word can contain either an immediate data word, a DCB address, or zeros. The contents of this word are device dependent. Refer to the device description manual for the meaning of the two words of IDCB information.

Printing or Displaying the Log Information *(continued)*

Item 14 shows the value of the interrupt condition code. The code indicates successful or unsuccessful completion of the I/O operation. The meaning of the interrupt condition code is device dependent. Refer to the device description manual for the meaning of this code.

Item 15 shows the value of the interrupt status byte (ISB). The ISB contains additional information about the I/O error. The meaning of the ISB is device dependent. Refer to the device description manual for the meaning of this value.

Item 16 shows the device control block (DCB) information for this device when the I/O error occurred. If the device did not require a DCB to perform the I/O operation, this item would not appear in the listing. This example shows the contents of three chained DCBs the device needed to perform the I/O.

Item 17 shows the contents of the cycle steal status words (CSSW) when the I/O error occurred. Each word provides some information about the error. The number of words varies by device type and in some cases by error type. Refer to the device description manual for the meaning of the cycle steal status words.

Item 18 shows information about the I/O error that occurred on the device at address X‘21’. Item 4 shows that only one I/O error occurred at this address.

The value X‘0306’ shown below item 19 means that this device is a 4973 printer.

Notice that for this device, no DCBs were required to do the I/O and that eight words of cycle steal status were logged.

Figure 34 shows the format of a log entry for an application program check and a system program check. Refer to "How to Interpret the Program Check Message" on page PD-44 for more information on the various fields shown in this example.

```
1                   ***  PROGRAM CHECK  ***
2  DATE:  11/15/85                  TIME:  08:25:  31
   3                4               5
   SAR = 904B       PSW = 8002      PSW ANALYSIS:  SPECIFICATION CHECK
                                                   TRANSLATOR ENABLED
6  ADDRESS OF TCB = 004C        7   PROGRAM NAME: TESTRUN
   8                9               10
   IAR = 2E7A       AKR = 0330      LSR = 00D0
   11
   R0 (WORK REG)      = 0000        R4 (EDL OP2 ADDR) = 1E45
   R1 (INSTR ADDR)    = 1E40        R5 (EDL COMMAND)  = 815C
   R2 (EDL TCB ADDR)  = 1E4C        R6 (WORK REG)     = 00B8
   R3 (EDL OP1 ADDR)  = 0000        R7 (WORK REG)     = 1E7A

12                  ***  SYSTEM CHECK  ***
   DATE:  11/15/85                  TIME:  08:25:31
   SAR = 90C2       PSW = 8002      PSW ANALYSIS:  SPECIFICATION CHECK
                                                   TRANSLATOR ENABLED
   ADDRESS OF TCB = 004C
   IAR = 7B20       AKR = 0300      LSR = 10D0
   R0 = 02BE                        R4 = 2222
   R1 = FFFF                        R5 = 7AFE
   R2 = 0904                        R6 = 8888
   R3 = A7A7                        R7 = 2222
```

Figure 34. Example of Program Check Log Entries

Item **1** indicates the type of program check information in the log record. $DISKUT2 indicates either an application program check (PROGRAM CHECK) or a system program check (SYSTEM CHECK).

Item **2** shows the date and time when the program check occurred, according to the system clock.

The contents of the Storage Address Register (SAR) are shown under item **3**. The SAR tells you which storage address the system was referring to when the program check occurred.

Item **4** shows the value of the processor status word (PSW) when the program check occurred. The PSW indicates the type of error encountered. The meaning of the PSW is shown under item **5**. In this example, a specification check occurred in the program. The **PSW ANALYSIS** field also shows that the Storage Address Relocation Translator Feature was installed and enabled.

Item **6** shows the address of the active task control block (TCB). The address is not relocated and reflects the address of the TCB in the program's compiler listing.

Item **7** shows the name of the failing program.

Printing or Displaying the Log Information *(continued)*

Item 8 shows the contents of the instruction address register (IAR). The address in the register is the address of the machine instruction that was executing when the program check occurred.

Item 9 shows the contents of the address key register (AKR). Item 10 shows the contents of the level status register.

The information under item 11 is a list of the general purpose registers (R0–R7) and their contents. For programs written in EDL, the contents of these registers are as follows:

Register	***Contents***
R0	Work register. The contents of this register are usually not significant.
R1	The address of the failing EDL instruction.
R2	The address in storage of the active task control block (TCB). The address in R2 is the sum of the TCB address and the load point address of the program.
R3	The address in storage of the operand 1 of the failing EDL instruction.
R4	The address in storage of operand 2 (if applicable) of the failing EDL instruction.
R5	The operation code of the failing EDL instruction.
R6	Work register. The contents of this register are usually not significant.
R7	Work register. The contents of this register are usually not significant.

Item 12 is a sample of a log record for a system program check. The format of the system program check is similar to that used for application program checks. Notice, however, that the general purpose registers are not labeled in the log entry for the system program check. The registers are not labeled because system program checks normally involve Series/1 assembler code where the contents of the registers can vary.

Appendix A. How to Use the Programmer Console

The programmer console, which is an optional Series/1 processor feature, is a useful tool when you analyze problems.

Several of the chapters in this book mention the use of the programmer console to display storage locations. However, you can perform many more functions with the programmer console. This appendix explains some additional functions you can do. You can use the programmer console to:

- Display or alter main storage locations
- Store data into main storage
- Display or alter register contents
- Store data into registers
- Stop on a selected address
- Stop on an error condition
- Execute one instruction at a time.

The topics discussed in this appendix use the term "console" when referring to the programmer console.

Before the various functions of the console are discussed, a section on how to read the indicator lights is presented. This section follows.

How to Use the Programmer Console

Reading the Console Indicator Lights

Across the top of the console is a row of 16 indicator lights. These lights represent the 16 binary bits of a Series/1 word or two bytes. You refer to each indicator light as a bit position. The bit positions are numbered left to right as bit position 0 through bit 15. When an indicator light is on, this means that that bit is on or set to 1.

The value displayed in the lights may represent data in storage or registers, or it may represent a storage address. What the value represents depends on the function you are performing. How the console represents a value and how you read that value is described as follows.

Each group of four binary indicators represents four bits of a word area. Byte 0 (group 1 and group 2) is the leftmost byte. Each light in a group of four has a binary-coded decimal value, as follows:

```
X X X X      X X X X      X X X X      X X X X
8 4 2 1      8 4 2 1      8 4 2 1      8 4 2 1

Group 1      Group 2      Group 3      Group 4
```

Figure 35. Indicator Lights — Example 1

If you add the values of any one group of four lights when each of the lights are on in that group, the total is 15 or F in hexadecimal.

Because data and addresses in the Series/1 are represented in hexadecimal, it is good practice to convert the binary-coded decimal values displayed by the lights to hexadecimal. Appendix C, "Conversion Table" on page PD-139 contains a table to help you convert from binary to hexadecimal.

In the following example, assume that the top row represents the indicator lights. The 0 represents lights that are off (set to 0) and X represents the lights that are on (set to 1).

```
0 0 0 X      0 0 X 0      0 X 0 X      X 0 0 0
      1          2          4   1      8

Group 1      Group 2      Group 3      Group 4
```

Figure 36. Indicator Lights — Example 2

In the second row is the decimal equivalent that corresponds to the X above the value. Add the values within each group of four to get the total value of each group. Thus, the value of the indicator lights in Figure 36 is 1 2 5 8.

Reading the Console Indicator Lights *(continued)*

Figure 37 shows a value which requires conversion to hexadecimal. The value of the indicator lights in this example is 1 3 9 A.

```
0 0 0 X     0 0 X X     X 0 0 X     X 0 X 0
      1         2 1     8     1     8   2

Group 1     Group 2     Group 3     Group 4
```

Figure 37. Indicator Lights — Example 3

The remaining sections explain the various functions of console.

How to Use the Programmer Console

Displaying Main Storage Locations

To display an area in main storage, do the following:

1. Press the Stop key.

2. Press the AKR (address key register) key. The contents of the AKR are displayed in the indicator lights.

3. Key in one hexadecimal value (new address key). This is the value of the address space (partition number minus 1) in which you want to display main storage. For example, to display main storage in partition 2, you would key in the value 1 on the console. The value you enter is displayed in bits 13–15 of the indicator lights.

4. Press the Store key to store the new address key into the AKR.

5. Press the SAR (storage address key) key. The contents of the SAR are displayed in the indicator lights.

6. Key in the address (four hexadecimal characters) you want to display. This address is displayed in the indicator lights.

7. Press the Store key. The address displayed in the lights is stored into the SAR.

8. Press the Main Storage key. The contents of storage at the address you entered is displayed in the indicator lights. To display sequential main storage locations, continue pressing the Main Storage key.

Each time you press the Main Storage key, the system increments the storage address by 2 and displays the contents at that address.

Storing Data into Main Storage

To store data area into main storage, do the following:

1. Press the Stop key.
2. Press the AKR (address key register) key. The contents of the AKR are displayed in the indicator lights.
3. Key in one hexadecimal value (new address key). This is the value of the address space (partition number minus 1) in which you want to store data. For example, to store data in partition 1, you would key in the value 0 on the console. The value you enter is displayed in bits 13–15 of the indicator lights.
4. Press the Store key to store the new address key into the AKR.
5. Press the SAR (storage address register) key. The contents of the SAR are displayed in the indicator lights.
6. Key in the address (four hexadecimal characters) at which you want to store data. The address you enter is displayed in the indicator lights.
7. Press the Store key. The address displayed in the indicator lights is stored into the SAR.
8. Press the Main Storage key. The contents of the address you entered is displayed in the indicator.
9. Key in the data (four hexadecimal digits) that you want stored at that address in main storage. The value you entered is displayed in the indicator lights.
10. Press the Store key. The value shown in the indicator lights is stored at the address you entered in step 6.

Each time you press the Store key, the system increments the SAR by 2, and the data stored at that location is displayed.

How to Use the Programmer Console

Displaying Register Contents

To display the contents of a register, do the following:

1. Press the Stop key.

2. Press the Level key for the hardware level that contains the register(s) you want to display. Timers run on level 0. The supervisor and attention list tasks run on level 1. User programs and tasks run on levels 2 and 3.

   You can display the contents of any of the following registers on that level by pressing the key for that register:

   LSR Level status register

   AKR Address key register

   IAR Instruction address register

   R0–R7 Hardware registers 0 through 7.

After you press the register key, the contents of that register are displayed in the indicator lights.

Storing Data into Registers

You can store data into the IAR or registers R0–R7 using the following procedure. The address key register (AKR) and level status register (LSR) are displayable only.

To store data into a register, do the following:

1. Press the Stop key.

2. Press the Level key for the hardware level that contains the register(s) in which you want to store data.

3. Press the key for the register in which the data is to be stored. The contents of that register are displayed in the indicator lights.

4. Key in the data that you want to store. The value you enter is displayed in the indicator lights.

5. Press the Store key. The value displayed in the indicator lights is stored in the register you selected.

Stopping at a Storage Address

To stop on an address, do the following:

1. Press the Stop key.

2. Press the AKR (address key register) key. The contents of the AKR are displayed in the indicator lights.

3. Key in one hexadecimal value (new address key). This is the value of the address space (partition number minus 1) which contains the address on which you want the system to stop. For example, to set a stop address in partition 1, you would key in the value 0 on the console. The value you enter is displayed in bits 13–15 of the indicator lights.

4. Press the Store key to store the new address key into the AKR.

5. Press the Stop On Address key.

6. Key in the address at which you want execution to stop.

7. Press the Store key. The address and address key are placed in the stop on address buffer.

8. Press the Start key. Execution begins at the current IAR address on the current hardware level.

When the system loads the address you specified into the IAR, the processor enters the stop state. At this point, you can examine the contents of storage. To exit the stop state, press the Start key; execution begins at the next sequential address.

Stopping When an Error Occurs

Pressing the Stop On Error key causes the system to stop immediately if it detects a program check, machine check, or power/thermal warning. To determine the error type, press the PSW (processor status word) key. The value of the PSW is displayed in the indicator lights. The section "Interpreting the Processor Status Word Bits" on page PD-47 explains what the bits indicate.

To restart the processor, press the Reset key then the Start key. Pressing only the Start key enables the processor to proceed with its error handling as if stop mode had not occurred.

How to Use the Programmer Console

Executing One Instruction at a Time

Pressing the Instruct Step key causes the system to execute one instruction and then stop.

To enable the system to execute one instruction at a time, do the following:

1. Press the Stop key.

2. Press the AKR (address key register) key. The contents of the AKR are displayed in the indicator lights.

3. Key in one hexadecimal value (new address key). This is the value of the address space (partition number minus 1) which contains the IAR address on which you want the system to stop. For example, if the IAR address was in partition 1, you would key in the value 0 on the console. The value you enter is displayed in bits 13–15 of the indicator lights.

4. Press the Store key to store the new address key into the AKR.

5. Press the Stop On Address key.

6. Key in the IAR address at which you want the system to stop.

7. Press the Store key. The IAR address and address key are placed in the stop on address buffer.

8. Press the Start key. When the system attempts to execute the IAR address, the processor stops.

9. Press the Instruct Step key. The system resets the Stop On Address to off.

10. Press the Start key. The system executes the instruction at the IAR address you entered and then stops. The system updates the IAR to point to the next instruction address.

Each time you press the Start key, one instruction is executed and the IAR is updated to the next instruction address.

If your supervisor contains timer support, interrupts will occur while you are single-instruction stepping through your program. When this happens, you enter the system interrupt handler at the time you press the Start key. You can set stop-on-address mode on your program's next instruction and press the Start key; then, single-step until the next interrupt.

If the processor is in run state, pressing the Instruct Step key causes the processor to enter the stop state. Pressing the Instruct Step key a second time resets instruction-step mode; the processor remains in the stop state.

Appendix B. Allowing IBM Access to Your System

On occasion, you may need to call an IBM support center to assist you in analyzing a problem with your system. If the problem is complex, the IBM support center representative may ask to establish a Remote Support Link. The Remote Support Link enables the support center representative to get direct access to your Series/1 system through a remote terminal. The link is established over a switched telephone line.

Using the Remote Support Link, an IBM support center representative can issue operator commands to your system and run EDX utilities. You can use the link to transfer disk data sets to the support center to assist representatives in diagnosing your problem.

This appendix describes the hardware you need to set up a Remote Support Link and the procedures for authorizing and disconnecting the link. To use these procedures, you must have defined a remote support terminal and included the necessary supervisor modules during system generation. See the *Installation and System Generation Guide* for more details.

You are responsible for ensuring the security and integrity of your data and software *before* giving IBM access to your system. You must, for example, give IBM permission to establish a Remote Support Link and should remove all confidential data from your system. IBM takes every precaution to ensure the integrity of your data and software, but IBM assumes no responsibility in this regard.

Allowing IBM Access to Your System

Hardware Requirements

To set up a Remote Support Link, you need the following hardware:

- One of the following communications adapters:
  - An Asynchronous Communications Single-Line Controller (#1610)
  - An Asynchronous Communications 8-Line Controller (#2091) with a 4-Line Adapter (#2092)
  - A Multifunction Attachment - Port 0 (#1310)
  - A Feature Programmable 8-Line Controller (#2095) with a 4-Line Adapter (#2096).
- A Communications Power Feature (#2010)
- An EIA[1] Communication Cable (#2057)
- A modem (compatible with the American Telephone & Telegraph Co. 212A modem)
- A voice-grade switched telephone line, preferably one that is not routed through a manually-operated switchboard.

  Note: It is easier for the IBM support center to assist you if you have a second telephone line available near your Series/1. The second line enables you to speak with a support center representative while your system is linked to the IBM support center.

In addition to the hardware just described, your system also must have a disk and diskette unit.

Authorizing the Link

If the IBM support center representative determines that a Remote Support Link would help in isolating or resolving your problem, you can use the following procedure to authorize the link. Remember, you are responsible for ensuring the security and integrity of your data and software *before* authorizing the link. You should, for example, remove all confidential data from your system.

1. Check to see that your modem is switched on and that the line is ready for use.

---

1 Electronic Industries Association

2. Load the IBM-supplied program called ANSWER. The ANSWER program resides on the IPL volume. You can load this program from any terminal and in any partition but all messages issued by the program appear on the operator console, $SYSLOG. To load the ANSWER program, press the attention key on your terminal and enter **$L ANSWER**

```
> $L ANSWER
LOADING ANSWER        3P,02:09:56, LP= 0000, PART= 2

IF YOU AGREE THAT IBM SHOULD INITIATE THE REMOTE SUPPORT LINK, AND
YOU HAVE TAKEN APPROPRIATE STEPS TO SAFEGUARD YOUR DATA, ENTER "Y"
    OR
ENTER  "N"  TO EXIT AND NOT ALLOW REMOTE SUPPORT ACCESS ==> _
```

3. If you have taken the appropriate steps to safeguard the data in your system, enter **Y** to authorize the Remote Support Link.

```
IF YOU AGREE THAT IBM SHOULD INITIATE THE REMOTE SUPPORT LINK, AND
YOU HAVE TAKEN APPROPRIATE STEPS TO SAFEGUARD YOUR DATA, ENTER "Y"
    OR
ENTER  "N"  TO EXIT AND NOT ALLOW REMOTE SUPPORT ACCESS ==> Y
NAME OF REMOTE TERMINAL ==> _
```

Entering **N** ends the program and prevents access to your system.

4. Enter the name of the remote support terminal. This name is the same as the label on the TERMINAL definition statement for the remote support terminal. In this example, the name of the terminal is REMSUPT.

```
IF YOU AGREE THAT IBM SHOULD INITIATE THE REMOTE SUPPORT LINK, AND
YOU HAVE TAKEN APPROPRIATE STEPS TO SAFEGUARD YOUR DATA, ENTER "Y"
    OR
ENTER  "N"  TO EXIT AND NOT ALLOW REMOTE SUPPORT ACCESS ==> Y
NAME OF REMOTE TERMINAL ==> REMSUPT
```

When you complete this step, the program enables the communications adapter and answers the phone when it receives a ring interrupt.

5. The IBM support center representative now has access to your Series/1 to diagnose a problem or to transfer a correction over the line. The support center representative can communicate with you by sending messages over the Remote Support Link or by talking with you on a separate telephone line.

Disconnecting the Link

To disconnect the line and end the ANSWER program, press the attention key and enter **HANGUP**.

```
> HANGUP

REMOTE SUPPORT LINE IS DISCONNECTED

ANSWER   ENDED AT 02:40:06
```

It is your responsibility to ensure that the Remote Support Link has been disconnected and disabled at the end of the problem-solving session.

Note: To communicate with your system, the IBM support center representative loads a program called RSLEDX1 from the remote terminal. If you disconnect the line before the support center representative ends RSLEDX1, the program will still be running on your system. You can cancel RSLEDX1 in this case by pressing the attention key and entering the $C command.

Appendix C. Conversion Table

This appendix contains a conversion table for the hexadecimal, binary, EDCBIC, and ASCII equivalents of decimal values. The table also contains transmission codes for communications devices.

Conversion Table

Decimal	Hex	Binary	EBCDIC	ASCII (see Notes 1 and 3)	EBASC* (see Notes 2 and 3)	EBCD	CRSP
0	00	0000 0000	NUL	NUL	NUL		
1	01	0001	SOH	SOH	NUL	space	space
2	02	0010	STX	STX	@	1	1,]
3	03	0011	ETX	ETX	@		
4	04	0100	PF	EOT	space	2	2
5	05	0101	HT	ENQ	space		
6	06	0110	LC	ACK	'		
7	07	0111	DEL	BEL	'	3	
8	08	1000		BS	DLE	4	5
9	09	1001	RLF	HT	DLE		
10	0A	1010	SMM	LF	P		
11	0B	1011	VT	VT	P	5	7
12	0C	1100	FF	FF	0		
13	0D	1101	CR	CR	0	6	6
14	0E	1110	SO	SO	p	7	8
15	0F	1111	SI	SI	p		
16	10	0001 0000	DLE	DLE	BS	8	4
17	11	0001	DC1	DC1	BS		
18	12	0010	DC2	DC2	H		
19	13	0011	TM	DC3	H	9	0
20	14	0100	RES	DC4	(		
21	15	0101	NL	NAK	(	0	Z
22	16	0110	BS	SYN	h	Ⓓ (EOA)	Ⓓ (EOA),9
23	17	0111	IL	ETB	h		
24	18	1000	CAN	CAN	CAN		
25	19	1001	EM	EM	CAN		
26	1A	1010	CC	SUB	X	RS	RS
27	1B	1011	CU1	ESC	X		
28	1C	1100	IFS	FS	8	upper case	upper case
29	ID	1101	IGS	GS	8		⩞
30	1E	1110	IRS	RS	x		
31	1F	1111	IUS	US	x	Ⓒ (EOT)	Ⓒ (EOT)
32	20	0010 0000	DS	space	EOT	@	t
33	21	0001	SOS	!	EOT		
34	22	0010	FS	"	D		
35	23	0011		#	D	/	x
36	24	0100	BYP	$	$		
37	25	0101	LF	%	$	s	n
38	26	0110	ETB	&	d	t	u
39	27	0111	ESC	'	d		
40	28	1000		(	DC4		
41	29	1001		)	DC4	u	e
42	2A	1010	SM	*	T	v	d
43	2B	1011	CU2	+	T		
44	2C	1100		,	4	w	k
45	2D	1101	ENQ	-	4		
46	2E	1110	ACK	.	t		
47	2F	1111	BEL	/	t	x	c
48	30	0011 0000		0	form feed		
49	31	0001		1	form feed	y	l
50	32	0010	SYN	2	L	z	h

*The no-parity TWX code for any given character is the code that has the rightmost bit position off.

Decimal	Hex	Binary	EBCDIC	ASCII (see Notes 1 and 3)	EBASC* (see Notes 2 and 3)	EBCD	CRSP
51	33	0011		3	L		
52	34	0100	PN	4	,		
53	35	0101	RS	5	,		
54	36	0110	UC	6	1	SOA	
55	37	0011 0111	EOT	7	1	Ⓢ (SOA),comma	b
56	38	1000		8	FS		
57	39	1001		9	FS		
58	3A	1010		:	\		
59	3B	1011	CU3	;	\	index	index
60	3C	1100	DC4	<	<		
61	3D	1101	NAK	=	<	Ⓑ (EOB)	
62	3E	1110		>	|		
63	3F	1111	SUB	?	|		
64	40	0100 0000	space	@	STX	Ⓝ (NAK),-	!
65	41	0001		A	STX		
66	42	0010		B	B		
67	43	0011		C	B	i	m
68	44	0100		D	''		
69	45	0101		E	''	k	
70	46	0110		F	b	l	v
71	47	0111		G	b		
72	48	1000		H	DC2		
73	49	1001		I	DC2	m	'
74	4A	1010	¢	J	R	n	r
75	4B	1011	.	K	R		
76	4C	1100	<	L	2	o	i
77	4D	1101	(	M	2		
78	4E	1110	+	N	r		
79	4F	1111	|	O	r	p	a
80	50	0101 0000	&	P	line feed		
81	51	0001		Q	line feed	q	o
82	52	0010		R	J	r	s
83	53	0011		S	J		
84	54	0100		T	*		
85	55	0101		U	*		
86	56	0110		V	j		
87	57	0111		W	j	$	w
88	58	1000		X	SUB		
89	59	1001		Y	SUB		
90	5A	1010	!	Z	Z		
91	5B	1011	$	[	Z	CRLF	CRLF
92	5C	1100	*	\	:		
93	5D	1101	)	]	:	backspace	backspace
94	5E	1110	;	∧	z	idle	idle
95	5F	1111	¬	—	z		
96	60	0110 0000	-	`	ACK		
97	61	0001	/	a	ACK	&	j
98	62	0010		b	F	a	g
99	63	0011		c	F		
100	64	0100		d	&	b	
101	65	0101		e	&		
102	66	0110		f	f		
103	67	0111		g	f	c	f

Conversion Table

Decimal	Hex	Binary	EBCDIC	ASCII (see Notes 1 and 3)	EBASC* (see Notes 2 and 3)	EBCD	CRSP
104	68	1000		h	SYN	d	p
105	69	1001		i	SYN		
106	6A	1010	¦	j	V		
107	6B	1011	,	k	V	e	
108	6C	1100	%	1	6		
109	6D	1101	_	m	6	f	q
110	6E	1110	>	n	v	g	comma
111	6F	1111	?	o	v		
112	70	0111 0000		p	shift out	h	/
113	71	0001		q	shift out		
114	72	0010		r	N		
115	73	0011		s	N	i	y
116	74	0100		t	.		
117	75	0101		u	.		
118	76	0110		v	n	Ⓨ (YAK),period	
119	77	0111		w	n		
120	78	1000		x	RS		
121	79	1001		y	RS		
122	7A	1010	:	z	∧	horiz tab	tab
123	7B	1011	#	{	∧		
124	7C	1100	@	¦	>	lower case	lower case
125	7D	1101	'	}	>		
126	7E	1110	=	~	∿		
127	7F	1111	"	DEL	∿	delete	
128	80	1000 0000		NUL	SOH		
129	81	0001	a	SOH	SOH	space	space
130	82	0010	b	STX	A	=	±, [
131	83	0011	c	ETX	A		
132	84	0100	d	EOT	!	<	@
133	85	0101	e	ENQ	!		
134	86	0110	f	ACK	a		
135	87	0111	g	BEL	a	;	#
136	88	1000	h	BS	DC1	:	%
137	89	1001	i	HT	DC1		
138	8A	1010		LF	Q		
139	8B	1011		VT	Q	%	&
140	8C	1100		FF	1		
141	8D	1101		CR	1	'	¢
142	8E	1110		SO	q	>	*
143	8F	1111		SI	q		
144	90	1001 0000		DLE	horiz tab	*	$
145	91	0001	j	DC1	horiz tab		
146	92	0010	k	DC2	I		
147	93	0011	l	DC3	I	(	)
148	94	0100	m	DC4	)		
149	95	0101	n	NAK	)	)	Z
150	96	0110	o	SYN	i	D (EOA),"	(
151	97	0111	p	ETB	i		
152	98	1000	q	CAN	EM		
153	99	1001	r	EM	EM		
154	9A	1010		SUB	Y		
155	9B	1011		ESC	Y		
156	9C	1100		FS	9	upper case	upper case

Decimal	Hex	Binary	EBCDIC	ASCII (see Notes 1 and 3)	EBASC* (see Notes 2 and 3)	EBCD	CRSP
157	9D	1101		GS	9		
158	9E	1110		RS	y		
159	9F	1111		US	y	C (EOT)	C (EOT)
160	A0	1010 0000		Space	ENQ	¢	T
161	A1	0001		!	ENQ		
162	A2	0010	s	"	E		
163	A3	0011	t	#	E	?	X
164	A4	0100	u	$	%		
165	A5	0101	v	%	%	S	N
166	A6	1010 0110	w	&	e	T	U
167	A7	0111	x	'	e		
168	A8	1000	y	(	NAK		
169	A9	1001	z	)	NAK	U	E
170	AA	1010		*	U	V	D
171	AB	1011		+	U		
172	AC	1100		,	5	W	K
173	AD	1101		-	5		
174	AE	1110		.	u		
175	AF	1111		/	u	X	C
176	B0	1011 0000		0	return		
177	B1	0001		1	return	Y	L
178	B2	0010		2	M	Z	H
179	B3	0011		3	M		
180	B4	0100		4	-		
181	B5	0101		5	-		
182	B6	0110		6	m		
183	B7	0111		7	m	Ⓢ (SOA),|	B
184	B8	1000		8	GS		
185	B9	1001		9	GS		
186	BA	1010		:	]		
187	BB	1011		;	]	index	index
188	BC	1100		<	::		
189	BD	1101		=	=	Ⓑ (EOB),ETB	
190	BE	1110		>	{		
191	BF	1111		?	{		
192	C0	1100 0000	{	@	ETX	Ⓝ (NAK),–	
193	C1	0001	A	A	ETX		
194	C2	0010	B	B	C		
195	C3	0011	C	C	C	J	M
196	C4	0100	D	D	#		
197	C5	0101	E	E	#	K	
198	C6	0 0110	F	F	c	L	V
199	C7	0111	G	G	c		
200	C8	1000	H	H	DC3		
201	C9	1001	I	I	DC3	M	"
202	CA	1010		J	S	N	R
203	CB	1011		K	S		
204	CC	1100	⑁	L	3	O	I
205	CD	1101		M	3		
206	CE	1110	⑂	N	s		
207	CF	1111		O	s	P	A
208	D0	1101 0000	}	P	vertical tab		
209	D1	0001	J	Q	vertical tab	Q	O

Conversion Table

Decimal	Hex	Binary	EBCDIC	ASCII (see Notes 1 and 3)	EBASC* (see Notes 2 and 3)	EBCD	CRSP
210	D2	0010	K	R	K	R	S
211	D3	0011	L	S	K		
212	D4	0100	M	T	+		
213	D5	0101	N	U	+		
214	D6	0110	O	V	k		
215	D7	0111	P	W	k	!	W
216	D8	1000	Q	X	ESC		
217	D9	1001	R	Y	ESC		
218	DA	1010		Z	[		
219	DB	1011		[	[	CRLF	CRLF
220	DC	1100		\	;		
221	DD	1101		]	;	backspace	backspace
222	DE	1110		Λ	}	idle	idle
223	DF	1111			}		
224	EO	1110 0000	\		bell		
225	E1	0001		a	bell	ı	J
226	E2	0010	S	b	G	A	G
227	E3	0011	T	c	G		
228	E4	0100	U	d	'	B	+
229	E5	0101	V	e	'		
230	E6	0110	W	f	g		
231	E7	0111	X	g	g	C	F
232	E8	1000	Y	h	ETB	D	P
233	E9	1001	Z	i	ETB		
234	EA	1010		j	W		
235	EB	1011		k	W	E	
236	EC	1100	⑁	l	7		
237	ED	1101		m	7	F	Q
238	EE	1110		n	w	G	comma
239	EF	1111		o	w		
240	FO	1111 0000	0	p	shift in	H	?
241	F1	0001	1	q	shift in		
242	F2	0010	2	r	O		
243	F3	0011	3	s	O	I	Y
244	F4	0100	4	t	/		
245	F5	0101	5	u	/		
246	F6	0110	6	v	o	Ⓨ (YAK), ¬	
247	F7	0111	7	w	o		
248	F8	1000	8	x	US		
249	F9	1001	9	y	US		
250	FA	1010	LVM	z	–	horiz tab	tab
251	FB	1011		{	–		
252	FC	1100		|	?	lower case	lower case
253	FD	1101		}	?		
254	FE	1110		~	DEL		
255	FF	1111		DEL	DEL	delete	

Notes:

1. ASCII terminals attached via #1310, #7850, #2095 with #2096, or #2095 with RPQ D02350.
2. ASCII terminals attached via #1610 or #2091 with #2092.
3. There are two entries for each character, depending on whether the parity is odd or even.

Glossary of Terms and Abbreviations

This glossary defines terms and abbreviations used in the Series/1 Event Driven Executive software publications. All software and hardware terms pertain to EDX. This glossary also serves as a supplement to the *IBM Data Processing Glossary*, GC20-1699.

$SYSLOGA, $SYSLOGB. The name of the alternate system logging device. This device is optional but, if defined, should be a terminal with keyboard capability, not just a printer.

$SYSLOG. The name of the system logging device or operator station; must be defined for every system. It should be a terminal with keyboard capability, not just a printer.

$SYSPRTR. The name of the system printer.

abend. Abnormal end-of-task. Termination of a task prior to its completion because of an error condition that cannot be resolved by recovery facilities while the task is executing.

ACCA. See asynchronous communications control adapter.

address key. Identifies a set of Series/1 segmentation registers and represents an address space. It is one less than the partition number.

address space. The logical storage identified by an address key. An address space is the storage for a partition.

application program manager. The component of the Multiple Terminal Manager that provides the program management facilities required to process user requests. It controls the contents of a program area and the execution of programs within the area.

application program stub. A collection of subroutines that are appended to a program by the linkage editor to provide the link from the application program to the Multiple Terminal Manager facilities.

asynchronous communications control adapter. An ASCII terminal attached via #1610, #2091 with #2092, or #2095 with #2096 adapters.

attention key. The key on the display terminal keyboard that, if pressed, tells the operating system that you are entering a command.

attention list. A series of pairs of 1 to 8 byte EBCDIC strings and addresses pointing to EDL instructions. When the attention key is pressed on the terminal, the operator can enter one of the strings to cause the associated EDL instructions to be executed.

backup. A copy of data to be used in the event the original data is lost or damaged.

base record slots. Space in an indexed file that is reserved for based records to be placed.

base records. Records are placed into an indexed file while in load mode or inserted in process mode with a new high key.

basic exchange format. A standard format for exchanging data on diskettes between systems or devices.

binary synchronous device data block (BSCDDB). A control block that provides the information to control one Series/1 Binary Synchronous Adapter. It determines the line characteristics and provides dedicated storage for that line.

Glossary of Terms and Abbreviations

block. (1) See data block or index block. (2) In the Indexed Method, the unit of space used by the access method to contain indexes and data.

block mode. The transmission mode in which the 3101 Display Station transmits a data data stream, which has been edited and stored, when the SEND key is pressed.

BSCAM. See binary synchronous communications access method.

binary synchronous communications access method. A form of binary synchronous I/O control used by the Series/1 to perform data communications between local or remote stations.

BSCDDB. See binary synchronous device data block.

buffer. An area of storage that is temporarily reserved for use in performing an input/output operation, into which data is read or from which data is written. See input buffer and output buffer.

bypass label processing. Access of a tape without any label processing support.

CCB. See terminal control block.

central buffer. The buffer used by the Indexed Access Method for all transfers of information between main storage and indexed files.

character image. An alphabetic, numeric, or special character defined for an IBM 4978 Display Station. Each character image is defined by a dot matrix that is coded into eight bytes.

character image table. An area containing the 256 character images that can be defined for an IBM 4978 Display Station. Each character image is coded into eight bytes, the entire table of codes requiring 2048 bytes of storage.

character mode. The transmission mode in which the 3101 Display Station immediately sends a character when a keyboard key is pressed.

cluster. In an indexed file, a group of data blocks that is pointed to from the same primary-level index block, and includes the primary-level index block. The data records and blocks contained in a cluster are logically contiguous, but are not necessarily physically contiguous.

COD (change of direction). A character used with ACCA terminal to indicate a reverse in the direction of data movement.

cold start. Starting the spool facility by erasing any spooled jobs remaining in the spool data set from any previous spool session.

command. A character string from a source external to the system that represents a request for action by the system.

common area. A user-defined data area that is mapped into the partitions specified on the SYSTEM definition statement. It can be used to contain control blocks or data that will be accessed by more than one program.

completion code. An indicator that reflects the status of the execution of a program. The completion code is displayed or printed on the program's output device.

constant. A value or address that remains unchanged thoughout program execution.

controller. A device that has the capability of configuring the GPIB bus by designating which devices are active, which devices are listeners, and which device is the talker. In Series/1 GPIB implementation, the Series/1 is always the controller.

conversion. See update.

control station. In BSCAM communications, the station that supervises a multipoint connection, and performs polling and selection of its tributary stations. The status of control station is assigned to a BSC line during system generation.

cross-partition service. A function that accesses data in two partitions.

cross-partition supervisor. A supervisor in which one or more supervisor modules reside outside of partition 1 (address space 0).

data block. In an indexed file, an area that contains control information and data records. These blocks are a multiple of 256 bytes.

data record. In an indexed file, the records containing customer data.

data set. A group of records within a volume pointed to by a directory member entry in the directory for the volume.

data set control block (DSCB). A control block that provides the information required to access a data set, volume or directory using READ and WRITE.

data set shut down. An indexed data set that has been marked (in main storage only) as unusable due to an error.

DCE. See directory control entry.

device data block (DDB). A control block that describes a disk or diskette volume.

direct access. (1) The access method used to READ or WRITE records on a disk or diskette device by specifying their location relative the beginning of the data set or volume. (2) In the Indexed Access Method, locating any record via its key without respect to the previous operation. (3) A condition in terminal I/O where a READTEXT or a PRINTEXT is directed to a buffer which was previously enqueued upon by an IOCB.

directory. (1) A series of contiguous records in a volume that describe the contents in terms of allocated data sets and free space. (2) A series of contiguous records on a device that describe the contents in terms of allocated volumes and free space. (3) For the Indexed Access Method Version 2, a data set that defines the relationship between primary and secondary indexed files (secondary index support).

directory control entry (DCE). The first 32 bytes of the first record of a directory in which a description of the directory is stored.

directory member entry (DME). A 32-byte directory entry describing an allocated data set or volume.

display station. An IBM 4978, 4979, or 3101 display terminal or similar terminal with a keyboard and a video display.

DME. See directory member entry.

DSCB. See data set control block.

dynamic storage. An increment of storage that is appended to a program when it is loaded.

end-of-data indicator. A code that signals that the last record of a data set has been read or written. End-of-data is determined by an end-of-data pointer in the DME or by the physical end of the data set.

ECB. See event control block.

EDL. See Event Driven Language.

emulator. The portion of the Event Driven Executive supervisor that interprets EDL instructions and performs the function specified by each EDL statement.

end-of-tape (EOT). A reflective marker placed near the end of a tape and sensed during output. The marker signals that the tape is nearly full.

enter key. The key on the display terminal keyboard that, if pressed, tells the operating system to read the information you entered.

event control block (ECB). A control block used to record the status (occurred or not occurred) of an event; often used to synchronize the execution of tasks. ECBs are used in conjunction with the WAIT and POST instructions.

Event Driven Language (EDL). The language for input to the Event Driven Executive compiler ($EDXASM), or the Macro and Host assemblers in conjunction with the Event Driven Executive macro libraries. The output is interpreted by the Event Driven Executive emulator.

EXIO (execute input or output). An EDL facility that provides user controlled access to Series/1 input/output devices.

external label. A label attached to the outside of a tape that identifies the tape visually. It usually contains items of identification such as file name and number, creation data, number of volumes, department number, and so on.

external name (EXTRN). The 1- to 8-character symbolic EBCDIC name for an entry point or data field that is not defined within the module that references the name.

FCA. See file control area.

FCB. See file control block.

file. A set of related records treated as a logical unit. Although file is often used interchangeably with data set, it usually refers to an indexed or a sequential data set.

file control area (FCA). A Multiple Terminal Manager data area that describes a file access request.

file control block (FCB). The first block of an indexed file. It contains descriptive information about the data contained in the file.

file control block extension. The second block of an indexed file. It contains the file definition parameters used to define the file.

file manager. A collection of subroutines contained within the program manager of the Multiple Terminal Manager that provides common support for all disk data transfer operations as needed for transaction-oriented application programs. It supports indexed and direct files under the control of a single callable function.

floating point. A positive or negative number that can have a decimal point.

formatted screen image. A collection of display elements or display groups (such as operator prompts and field input names and areas) that are presented together at one time on a display device.

free pool. In an indexed data set, a group of blocks that can be used for either data blocks or index blocks. These differ from other free blocks in that these are not initially assigned to specific logical positions in the file.

free space. In an indexed file, records blocks that do not currently contain data, and are available for use.

free space entry (FSE). An 8-byte directory entry defining an area of free space within a volume or a device.

FSE. See free space entry.

general purpose interface bus. The IEEE Standard 488-1975 that allows various interconnected devices to be attached to the GPIB adapter (RPQ D02118).

Glossary of Terms and Abbreviations

GPIB. See general purpose interface bus.

group. A unit of 100 records in the spool data set allocated to a spool job.

H exchange format. A standard format for exchanging data on diskettes between systems or devices.

host assembler. The assembler licensed program that executes in a 370 (host) system and produces object output for the Series/1. The source input to the host assembler is coded in Event Driven Language or Series/1 assembler language. The host assembler refers to the System/370 Program Preparation Facility (5798-NNQ).

host system. Any system whose resources are used to perform services such as program preparation for a Series/1. It can be connected to a Series/1 by a communications link.

IACB. See indexed access control block.

IAR. See instruction address register.

ICB. See indexed access control block.

IIB. See interrupt information byte.

image store. The area in a 4978 that contains the character image table.

immediate data. A self-defining term used as the operand of an instruction. It consists of numbers, messages or values which are processed directly by the computer and which do not serve as addresses or pointers to other data in storage.

index. In an indexed file, an ordered collection of pairs of keys and pointers, used to sequence and locate records.

index block. In an indexed file, an area that contains control information and index entries. These blocks are a multiple of 256 bytes.

indexed access control block (IACB/ICB). The control block that relates an application program to an indexed file.

indexed access method. An access method for direct or sequential processing of fixed-length records by use of a record's key.

indexed data set. Synonym for indexed file.

indexed file. A file specifically created, formatted and used by the Indexed Access Method. An indexed file is sometimes called an indexed data set.

index entry. In an indexed file, a key-pointer pair, where the pointer is used to locate a lower-level index block or a data block.

index register (#1, #2). Two words defined in EDL and contained in the task control block for each task. They are used to contain data or for address computation.

input buffer. (1) See buffer. (2) In the Multiple Terminal Manager, an area for terminal input and output.

input output control block (IOCB). A control block containing information about a terminal such as the symbolic name, size and shape of screen, the size of the forms in a printer, or an optional reference to a user provided buffer.

instruction address register (IAR). The pointer that identifies the machine instruction currently being executed. The Series/1 maintains a hardware IAR to determine the Series/1 assembler instruction being executed. It is located in the level status block (LSB).

integer. A positive or negative number that has no decimal point.

interactive. The mode in which a program conducts a continuous dialogue between the user and the system.

internal label. An area on tape used to record identifying information (similar to the identifying information placed on an external label). Internal labels are checked by the system to ensure that the correct volume is mounted.

interrupt information byte (IIB). In the Multiple Terminal Manager, a word containing the status of a previous input/output request to or from a terminal.

invoke. To load and activate a program, utility, procedure, or subroutine into storage so it can run.

job. A collection of related program execution requests presented in the form of job control statements, identified to the jobstream processor by a JOB statement.

job control statement. A statement in a job that specifies requests for program execution, program parameters, data set definitions, sequence of execution, and, in general, describes the environment required to execute the program.

job stream processor. The job processing facility that reads job control statements and processes the requests made by these statements. The Event Driven Executive job stream processor is $JOBUTIL.

jumper. (1) A wire or pair of wires which are used for the arbitrary connection between two circuits or pins in an attachment card. (2) To connect wire(s) to an attachment card or to connect two circuits.

key. In the Indexed Access Method, one or more consecutive characters used to identify a record and establish its order with respect to other records. See also key field.

key field. A field, located in the same position in each record of an indexed file, whose content is used for the key of a record.

level status block (LSB). A Series/1 hardware data area that contains processor status. This area is eleven words in length.

library. A set of contiguous records within a volume. It contains a directory, data sets and/or available space.

line. A string of characters accepted by the system as a single input from a terminal; for example, all characters entered before the carriage return on the teletypewriter or the ENTER key on the display station is pressed.

link edit. The process of resolving external symbols in one or more object modules. A link edit is performed with $EDXLINK whose output is a loadable program.

listener. A controller or active device on a GPIB bus that is configured to accept information from the bus.

load mode. In the Indexed Access Method, the mode in which records are loaded into base record slots in an indexed file.

load module. A single module having cross references resolved and prepared for loading into storage for execution. The module is the output of the $UPDATE or $UPDATEH utility.

load point. (1) Address in the partition where a program is loaded. (2) A reflective marker placed near the beginning of a tape to indicate where the first record is written.

lock. In the Indexed Access Method, a method of indicating that a record or block is in use and is not available for another request.

logical screen. A screen defined by margin settings, such as the TOPM, BOTM, LEFTM and RIGHTM parameters of the TERMINAL or IOCB statement.

LSB. See level status block.

mapped storage. The processor storage that you defined on the SYSTEM statement during system generation.

member. A term used to identify a named portion of a partitioned data set (PDS). Sometimes member is also used as a synonym for a data set. See data set.

menu. A formatted screen image containing a list of options. The user selects an option to invoke a program.

menu-driven. The mode of processing in which input consists of the responses to prompting from an option menu.

message. In data communications, the data sent from one station to another in a single transmission. Stations communication with a series of exchanged messages.

multifile volume. A unit of recording media, such as tape reel or disk pack, that contains more than one data file.

multiple terminal manager. An Event Driven Executive licensed program that provides support for transaction-oriented applications on a Series/1. It provides the capability to define transactions and manage the programs that support those transactions. It also manages multiple terminals as needed to support these transactions.

multivolume file. A data file that, due to its size, requires more than one unit of recording media (such as tape reel or disk pack) to contain the entire file.

new high key. A key higher than any other key in an indexed file.

nonlabeled tapes. Tapes that do not contain identifying labels (as in standard labeled tapes) and contain only files separated by tapemarks.

null character. A user-defined character used to define the unprotected fields of a formatted screen.

option selection menu. A full screen display used by the Session Manager to point to other menus or system functions, one of which is to be selected by the operator. (See primary option menu and secondary option menu.)

output buffer. (1) See buffer. (2) In the Multiple Terminal Manager, an area used for screen output and to pass data to subsequent transaction programs.

overlay. The technique of reusing a single storage area allocated to a program during execution. The storage area can be reused by loading it with overlay programs that have been specified in the PROGRAM statement of the program or by calling overlay segments that have been specified in the OVERLAY statement of $EDXLINK.

overlay area. A storage area within a program reserved for overlay programs specified in the PROGRAM statement or overlay segments specified in the OVERLAY statement in $EDXLINK.

overlay program. A program in which certain control sections can use the same storage location at different times during execution. An overlay program can execute concurrently as an asynchronous task with other programs and is specified in the EDL PROGRAM statement in the main program.

overlay segment. A self-contained portion of a program that is called and sequentially executes as a synchronous task. The entire program that calls the overlay segment need not be maintained in storage while the overlay segment is executing. An overlay segment is specified in the OVERLAY statement of $EDXLINK or $XPSLINK (for initialization modules).

overlay segment area. A storage area within a program or supervisor reserved for overlay segments. An overlay segment area is specified with the OVLAREA statement of $EDXLINK.

Glossary of Terms and Abbreviations

parameter selection menu. A full screen display used by the Session Manager to indicate the parameters to be passed to a program.

partition. A contiguous fixed-sized area of storage. Each partition is a separate address space.

performance volume. A volume whose name is specified on the DISK definition statement so that its address is found during IPL, increasing system performance when a program accesses the volume.

physical timer. Synonym for timer (hardware).

polling. In data communications, the process by which a multipoint control station asks a tributary if it can receive messages.

precision. The number of words in storage needed to contain a value in an operation.

prefind. To locate the data sets or overlay programs to be used by a program and to store the necessary information so that the time required to load the prefound items is reduced.

primary file. An indexed file containing the data records and primary index.

primary file entry. For the Indexed Access Method Version 2, an entry in the directory describing a primary file.

primary index. The index portion of a primary file. This is used to access data records when the primary key is specified.

primary key. In an indexed file, the key used to uniquely identify a data record.

primary-level index block. In an indexed file, the lowest level index block. It contains the relative block numbers (RBNs) and high keys of several data blocks. See cluster.

primary menu. The program selection screen displayed by the Multiple Terminal Manager.

primary option menu. The first full screen display provided by the Session Manager.

primary station. In a Series/1 to Series/1 attachment, the processor that control communication between the two computers. Contrast with secondary station.

primary task. The first task executed by the supervisor when a program is loaded into storage. It is identified by the PROGRAM statement.

priority. A combination of hardware interrupt level priority and a software ranking within a level. Both primary and secondary tasks will execute asynchronously within the system according to the priority assigned to them.

process mode. In the Indexed Access Method, the mode in which records can be retrieved, updated, inserted or deleted.

processor status word (PSW). A 16-bit register used to (1) record error or exception conditions that may prevent further processing and (2) hold certain flags that aid in error recovery.

program. A disk- or diskette-resident collection of one or more tasks defined by a PROGRAM statement; the unit that is loaded into storage. (See primary task and secondary task.)

program header. The control block found at the beginning of a program that identifies the primary task, data sets, storage requirements and other resources required by a program.

program/storage manager. A component of the Multiple Terminal Manager that controls the execution and flow of application programs within a single program area and contains the support needed to allow multiple operations and sharing of the program area.

protected field. A field in which the operator cannot use the keyboard to enter, modify, or erase data.

PSW. See processor status word.

QCB. See queue control block.

QD. See queue descriptor.

QE. See queue element.

queue control block (QCB). A data area used to serialize access to resources that cannot be shared. See serially reusable resource.

queue descriptor (QD). A control block describing a queue built by the DEFINEQ instruction.

queue element (QE). An entry in the queue defined by the queue descriptor.

quiesce. To bring a device or a system to a halt by rejection of new requests for work.

quiesce protocol. A method of communication in one direction at a time. When sending node wants to receive, it releases the other node from its quiesced state.

record. (1) The smallest unit of direct access storage that can be accessed by an application program on a disk or diskette using READ and WRITE. Records are 256 bytes in length. (2) In the Indexed Access Method, the logical unit that is transferred between $IAM and the user's buffer. The length of the buffer is defined by the user. (3) In BSCAM communications, the portions of data transmitted in a message. Record length (and, therefore, message length) can be variable.

recovery. The use of backup data to re-create data that has been lost or damaged.

reflective marker. A small adhesive marker attached to the reverse (nonrecording) surface of a reel of magnetic tape. Normally, two reflective markers are used on each reel of tape. One indicates the beginning of the recording area on the tape (load point), and the other indicates the proximity to the end of the recording area (EOT) on the reel.

relative block address (RBA). The location of a block of data on a 4967 disk relative to the start of the device.

relative record number. An integer value identifying the position of a record in a data set relative to the beginning of the data set. The first record of a data set is record one, the second is record two, the third is record three.

relocation dictionary (RLD). The part of an object module or load module that is used to identify address and name constants that must be adjusted by the relocating loader.

remote management utility control block (RCB). A control block that provides information for the execution of remote management utility functions.

reorganize. The process of copying the data in an indexed file to another indexed file in a manner that rearranges the data for more optimum processing and free space distribution.

restart. Starting the spool facility w the spool data set contains jobs from a previous session. The jobs in the spool data set can be either deleted or printed when the spool facility is restarted.

return code. An indicator that reflects the results of the execution of an instruction or subroutine. The return code is usually placed in the task code word (at the beginning of the task control block).

roll screen. A display screen which is logically segmented into an optional history area and a work area. Output directed to the screen starts display at the beginning of the work area and continues on down in a line-by-line sequence. When the work area gets full, the operator presses ENTER/SEND and its contents are shifted into the optional history area and the work area itself is erased. Output now starts again at the beginning of the work area.

SBIOCB. See sensor based I/O control block.

second-level index block. In an indexed data set, the second-lowest level index block. It contains the addresses and high keys of several primary-level index blocks.

secondary file. See secondary index.

secondary index. For the Indexed Access Method Version 2, an indexed file used to access data records by their secondary keys. Sometimes called a secondary file.

secondary index entry. For the Indexed Access Method Version 2, this an an entry in the directory describing a secondary index.

secondary key. For the Indexed Access Method Version 2, the key used to uniquely identify a data record.

secondary option menu. In the Session Manager, the second in a series of predefined procedures grouped together in a hierarchical structure of menus. Secondary option menus provide a breakdown of the functions available under the session manager as specified on the primary option menu.

secondary task. Any task other than the primary task. A secondary task must be attached by a primary task or another secondary task.

secondary station. In a Series/1 to Series/1 attachment, the processor that is under the control of the primary station.

sector. The smallest addressable unit of storage on a disk or diskette. A sector on a 4962 or 4963 disk is equivalent to an Event Driven Executive record. On a 4964 or 4966 diskette, two sectors are equivalent to an Event Driven Executive record.

selection. In data communications, the process by which the multipoint control station asks a tributary station if it is ready to send messages.

self-defining term. A decimal, integer, or character that the computer treats as a decimal, integer, or character and not as an address or pointer to data in storage.

sensor based I/O control block (SBIOCB). A control block containing information related to sensor I/O operations.

sequential access. The processing of a data set in order of occurrence of the records in the data set. (1) In the Indexed Access Method, the processing of records in ascending collating sequence order of the keys. (2) When using READ/WRITE, the processing of records in ascending relative record number sequence.

serially reusable resource (SRR). A resource that can only be accessed by one task at a time. Serially reusable resources are usually managed via (1) a QCB and ENQ/DEQ statements or (2) an ECB and WAIT/POST statements.

service request. A device generated signal used to inform the GPIB controller that service is required by the issuing device.

session manager. A series of predefined procedures grouped together as a hierarchical structure of menus from which you select the utility functions, program preparation facilities, and language processors needed to prepare and execute application programs. The menus consist of a primary option menu that displays functional groupings and secondary option menus that display a breakdown of these functional groupings.

shared resource. A resource that can be used by more than one task at the same time.

Glossary of Terms and Abbreviations

shut down. See data set shut down.

source module/program. A collection of instructions and statements that constitute the input to a compiler or assembler. Statements may be created or modified using one of the text editing facilities.

spool job. The set of print records generated by a program (including any overlays) while engueued to a printer designated as a spool device.

spool session. An invocation and termination of the spool facility.

spooling. The reading of input data streams and the writing of output data streams on storage devices, concurrently with job execution, in a format convenient for later processing or output operations.

SRQ. See service request.

stand-alone dump. An image of processor storage written to a diskette.

stand-alone dump diskette. A diskette supplied by IBM or created by the $DASDI utility.

standard labels. Fixed length 80-character records on tape containing specific fields of information (a volume label identifying the tape volume, a header label preceding the data records, and a trailer label following the data records).

static screen. A display screen formatted with predetermined protected and unprotected areas. Areas defined as operator prompts or input field names are protected to prevent accidental overlay by input data. Areas defined as input areas are not protected and are usually filled in by an operator. The entire screen is treated as a page of information.

station. In BSCAM communications, a BSC line attached to the Series/1 and functioning in a point-to-point or multipoint connection. Also, any other terminal or processor with which the Series/1 communicates.

subroutine. A sequence of instructions that may be accessed from one or more points in a program.

supervisor. The component of the Event Driven Executive capable of controlling execution of both system and application programs.

system configuration. The process of defining devices and features attached to the Series/1.

SYSGEN. See system generation.

system generation. The processing of defining I/O devices and selecting software options to create a supervisor tailored to the needs of a specific Series/1 hardware configuration and application.

system partition. The partition that contains the root segment of the supervisor (partition number 1, address space 0).

talker. A controller or active device on a GPIB bus that is configured to be the source of information (the sender) on the bus.

tape device data block (TDB). A resident supervisor control block which describes a tape volume.

tapemark. A control character recorded on tape used to separate files.

task. The basic executable unit of work for the supervisor. Each task is assigned its own priority and processor time is allocated according to this priority. Tasks run independently of each other and compete for the system resources. The first task of a program is the primary task. All tasks attached by the primary task are secondary tasks.

task code word. The first two words (32 bits) of a task's TCB; used by the emulator to pass information from system to task regarding the outcome of various operations, such as event completion or arithmetic operations.

task control block (TCB). A control block that contains information for a task. The information consists of pointers, save areas, work areas, and indicators required by the supervisor for controlling execution of a task.

task supervisor. The portion of the Event Driven Executive that manages the dispatching and switching of tasks.

TCB. See task control block.

terminal. A physical device defined to the EDX system using the TERMINAL configuration statement. EDX terminals include directly attached IBM displays, printers and devices that communicate with the Series/1 in an asynchronous manner.

terminal control block (CCB). A control block that defines the device characteristics, provides temporary storage, and contains links to other system control blocks for a particular terminal.

terminal environment block (TEB). A control block that contains information on a terminal's attributes and the program manager operating under the Multiple Terminal Manager. It is used for processing requests between the terminal servers and the program manager.

terminal screen manager. The component of the Multiple Terminal Manager that controls the presentation of screens and communications between terminals and transaction programs.

terminal server. A group of programs that perform all the input/output and interrupt handling functions for terminal devices under control of the Multiple Terminal Manager.

terminal support. The support provided by EDX to manage and control terminals. See terminal.

timer. The timer features available with the Series/1 processors. Specifically, the 7840 Timer Feature card (4955 only) or the native timer (4952, 4954, and 4956). Only one or the other is supported by the Event Driven Executive.

trace range. A specified number of instruction addresses within which the flow of execution can be traced.

transaction oriented applications. Program execution driven by operator actions, such as responses to prompts from the system. Specifically, applications executed under control of the Multiple Terminal Manager.

transaction program. See transaction-oriented applications.

transaction selection menu. A Multiple Terminal Manager display screen (menu) offering the user a choice of functions, such as reading from a data file, displaying data on a terminal, or waiting for a response. Based upon the choice of option, the application program performs the requested processing operation.

tributary station. In BSCAM communications, the stations under the supervision of a control station in a multipoint connection. They respond to the control station's polling and selection.

unmapped storage. The processor storage in your processor that you did not define on the SYSTEM statement during system generation.

unprotected field. A field in which the operator can use the keyboard to enter, modify or erase data. Also called non-protected field.

update. (1) To alter the contents of storage or a data set. (2) To convert object modules, produced as the output of an assembly or compilation, or the output of the linkage editor, into a form that can be loaded into storage for program execution and to update the directory of the volume on which the loadable program is stored.

user exit. (1) Assembly language instructions included as part of an EDL program and invoked via the USER instruction. (2) A point in an IBM-supplied program where a user written routine can be given control.

variable. An area in storage, referred to by a label, that can contain any value during program execution.

vary offline. (1) To change the status of a device from online to offline. When a device is offline, no data set can be accessed on that device. (2) To place a disk or diskette in a state where it is unknown by the system.

vary online. To place a device in a state where it is available for use by the system.

vector. An ordered set or string of numbers.

volume. A disk, diskette, or tape subdivision defined using $INITDSK or $TAPEUT1.

volume descriptor entry (VDE). A resident supervisor control block that describes a volume on a disk or diskette.

volume label. A label that uniquely identifies a single unit of storage media.

Index

The following index contains entries for this book only. See the *Library Guide and Common Index* for a Common Index to all Event Driven Executive books.

Index

L

M

N

O

P

Index

T

U

W

IBM Series/1 Event Driven Executive

Publications Order Form

Instructions:

1. Complete the order form, supplying all of the requested information. (Please print or type.)
2. If you are placing the order by phone, dial **1-800-IBM-2468.**
3. If you are mailing your order, fold the order form as indicated, seal with tape, and mail. We pay the postage.

Ship to:

Name: ____________________

____

Address: ____________________

____

City: ____________________

State: __________ Zip: __________

Bill to:

Customer number: ____________________

Name: ____________________

____

Address: ____________________

____

City: ____________________

State: __________ Zip: __________

Your Purchase Order No.: ____________________

Phone: (   ) ____________________

Signature: ____________________

Date: ____________________

Order:

Description	Order number	Qty.
Reference books:		
Set of the following six books. To order individual copies, use the following order numbers.	SBOF-1627	______
Communications Guide	SC34-0638	______
Extended Address Mode and Performance Analyzer User Guide	SC34-0591	______
Installation and System Generation Guide	SC34-0646	______
Language Reference	SC34-0643	______
Library Guide and Common Index	SC34-0645	______
Messages and Codes	SC34-0636	______
Operator Commands and Utilities Reference	SC34-0644	______
Guides and reference cards:		
Set of the following four books and reference cards. To order individual copies, use the following order numbers.	SBOF-1628	______
Customization Guide	SC34-0635	______
Event Driven Language Programming Guide	SC34-0637	______
Operation Guide	SC34-0642	______
Problem Determination Guide	SC34-0639	______
Language Reference Card	SX34-0165	______
Operator Commands and Utilities Reference Card	SX34-0164	______
Conversion Charts Reference Card	SX34-0163	______
Reference Card Envelope	SX34-0166	______
Set of three reference cards and storage envelope. (One set is included with order number SBOF-1627.)	SBOF-1629	______
Binders:		
3-ring easel binder with 1 inch rings	SR30-0324	______
3-ring easel binder with 2 inch rings	SR30-0327	______
Standard 3-ring binder with 1 inch rings	SR30-0329	______
Standard 3-ring binder with 1 1/2 inch rings	SR30-0330	______
Standard 3-ring binder with 2 inch rings	SR30-0331	______
Diskette binder (Holds eight 8-inch diskettes.)	SB30-0479	______

Publications Order Form

Cut or Fold Along Line

Fold and tape | Please Do Not Staple | Fold and tape

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

IBM Corporation
1 Culver Road
Dayton, New Jersey 08810

Fold and tape | Please Do Not Staple | Fold and tape

IBM®

International Business Machines Corporation

IBM Series/1 Event Driven Executive
Problem Determination Guide

Order No. SC34-0639-0

READER'S COMMENT FORM

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Note: Staples can cause problems with automated mail sorting equipment.
Please use pressure sensitive or other gummed tape to seal this form.

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

SC34-0639-0

Printed in U.S.A.

Reader's Comment Form

Cut or Fold Along Line

Fold and tape | Please Do Not Staple | Fold and tape

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Information Development, Department 28B
P.O. Box 1328
Boca Raton, Florida 33432

Fold and tape | Please Do Not Staple | Fold and tape

IBM®

IBM Series/1 Event Driven Executive
Problem Determination Guide

Order No. SC34-0639-0

READER'S COMMENT FORM

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Note: Staples can cause problems with automated mail sorting equipment.
Please use pressure sensitive or other gummed tape to seal this form.

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

SC34-0639-0
Printed in U.S.A.

Reader's Comment Form

Fold and tape | Please Do Not Staple | Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Information Development, Department 28B
P.O. Box 1328
Boca Raton, Florida 33432

Fold and tape | Please Do Not Staple | Fold and tape

IBM®

Cut or Fold Along Line

International Business Machines Corporation

SC34-0639-0

SC34-0639-0
Program Number: 5719-XS5
File Number: S1-37
Printed in U.S.A.